UNIVERSIDAD TECNOLÓGICA NACIONAL

Facultad Regional Santa Fe

Doctorado en Ingeniera

Mención en Ingeniera en Sistemas de Información

Tesis Doctoral

"DE^{2}M: Una Arquitectura para el Diseño y Simulación de Procesos de Empresa"

Ing. María de los Milagros Gutiérrez

Director: Dr. Horacio P. Leone

Santa Fe, Argentina. Diciembre de 2009

Gutiérrez, María de los Milagros

DE²M: Una arquitectura para el diseño y simulación de procesos de empresa.

1ª ed. Santa Fe 2010.

363 p., 29x21cm.

ISBN =78-987-05-8226-7

Sistemas de información.

CDD =58.406

Fecha de catalogación: 08/03/2010

Universidad Tecnológica Nacional

Facultad Regional Santa Fe

Comisión de Posgrado

Se presenta esta Tesis en cumplimiento de los requisitos exigidos por la Universidad Tecnológica Nacional para la obtención del grado académico de Doctor en Ingeniería, mención Sistemas de Información

"DE^{2}M: Una Arquitectura para el Diseño y Simulación de Procesos de Empresa"

por

Ing. María de los Milagros Gutiérrez

Director: Dr. Horacio P. Leone

Jurados de Tesis:

Dr. Carlos G. García Garino

Dr. Javier Orozco

Dra. Ana Rosa Tymoschuk

Santa Fe, Argentina. Diciembre de 2009

A Marcelo

A mis hijas: Milagros y Consuelo

Agradecimientos

A mi Dios... por el milagro de la vida y por los sueños.

A mi familia: Mili, Consu y Marce por su amor incondicional y el apoyo que siempre me dieron, sin el cual no hubiera podido seguir adelante.

A mi director Horacio, por la confianza y el aliento brindado; por sus buenos consejos, experiencias y conocimiento compartidos.

A los doctores Pablo, Laura, Mercedes, Georgina, Luciana por escucharme, alentarme y aconsejarme en esos momentos difíciles, transmitiéndome confianza, seguridad y tranquilidad.

A Georgina mi colaboradora, amiga y consejera por haberme enseñado a confiar en mi misma, por haber compartido sus experiencias, por impulsarme siempre a continuar y seguir adelante con nuevos proyectos.

Al grupo de las Brujas por esas lindas noches de Jueves de peña en Puerto café, donde dejábamos de lado la profesión para jugar el rol de madre, esposa, novia, chef, ama de casa, amigas...

A los que compartieron conmigo el lugar de trabajo: Mariela, Mariano, Geor, Jorge, Ivana, Mariel, Luciana, Juan Carlos, Merce, Laura, Pablo por los mates compartidos y

las largas charlas en los pasillos donde escucharon con tanto interés mis problemas, mis alegrías y mis logros.

A los que me enseñaron a usar Latex... Georgina, Luciana, Milton por compartir su conocimiento.

A mis becarios: Darío, Florencia, Milton y Jorge por enseñarme y ayudarme a resolver los problemas.

A los directivos de Andresito, por su amable y cordial recibimiento.

A los que perdí en el camino: Ricardo, David, Beba, Yoli por haberme dejado en mis recuerdos tantos momentos lindos compartidos con ustedes.

A mis hermanos y cuñados: Martín, Gaby, Fer, Mariana, Raúl, Silvina, Claudia, Victoria, Tato, Anibal y Gordo por esos asaditos de los domingos donde compartimos nuestra vida cotidiana.

A mis sobrinos queridos por alegrar mis días.

A la memoria de mis Padres por ser los instrumentos del milagro de la vida, por sus enseñanzas y cariño.

A mis amigas Lucila y Gabriela por compartir conmigo tantos momentos, por sus risas y lágrimas, por las caminatas en la costanera y las largas charlas llenas de cariñosos consejos.

A todos, Gracias por ser parte de mis días.

Resumen

Esta tesis, presenta la arquitectura de un entorno computacional para el diseño y ejecución de modelos de empresa ejecutables y distribuidos. Después de una breve introducción sobre el estado del arte y la presentación de la problemática, se abordar el concepto de Modelos de Empresa Ejecutables y Distribuidos y la propuesta para lograr obtener tales modelos con la arquitectura. Así, se hace una introducción al mundo de los modelos de empresa para conocer el por qué y el cómo de los mismos. Presentando la necesidad de un lenguaje de modelado de empresa con el cual plasmar el conocimiento de la misma a través de diferentes vistas, cada una conformada por un conjunto de diagramas. Más adelante se hace referencia al significado de ejecutar ese modelo de empresa a través del uso de simulación. La propuesta presenta una estrategia novedosa en la ejecución de los modelos, no solo por el paradigma usado sino también por los conceptos de reuso y modularidad que se tuvieron en cuenta. La ejecución de los modelos se presenta en dos entornos diferentes: local y distribuido, permitiendo analizar desde diferentes dimensiones los modelos de empresa. Finalmente se desarrolla un ejemplo que describe en detalle el uso de la arquitectura.

Índice general

Índice de Tablas

Índice de Figuras

1 Introducción

En este capítulo se realiza una introducción de la problemática que esta tesis aborda haciendo una revisión de la bibliografía, se plantean los objetivos que se persiguen, los principales aportes y por último se presenta la organización de la misma. Principalmente se hace referencia a la necesidad de las empresas de analizar sus procesos para poder adaptarse a los cambios continuos. Las diferentes estructuras que una empresa puede adoptar influyen sobre la solución de análisis que se plantea.

1.1. Contexto

Hacia los años '90 las compañías han hecho esfuerzos para organizar sus procesos de negocios internos, identificando sus actividades, principalmente aquellas relacionadas a la cadena de valor de sus productos. La economía cambiante, la globalización, la evolución de la tecnología, la personalización de los productos o servicios, imprimen en la empresa la necesidad de tomar ventajas de los entornos más difíciles, mejorar la calidad del producto, acercarse más al cliente, resaltando el mercado competitivo. Las mejores empresas son aquellas que aplican una política agresiva, que les permite crecer (desarrollar capacidades que asistan al cliente, cambiar costo por cantidad de clientes, atacar nuevos mercados, etc.). Es a partir de entonces que se produce una transformación en

la forma de entender a la empresa: deja de verse como una jerarquía en estructura y control (grande y rígida), para ser vista como un conjunto de unidades funcionales que se comunican unas con otras y que cooperan para lograr los objetivos organizacionales. Esta visión integrada de empresa se orienta a mejorar la performance de mercados y organizaciones distribuidas, localizándose en la comunicación de la información y la coordinación y optimización de las decisiones y los procesos. Se pretende alcanzar altos niveles de productividad, flexibilidad y calidad. Para lograrlo, las unidades de la empresa, ya sean estas humanas o máquinas, deben entenderse unas con otras, implicando la necesidad de un lenguaje común que pueda ser utilizado para construir un modelo de empresa (EM) que plasme el conocimiento que se tiene de la misma (Fox y Grüninger, 1997)(Fox y otros, 1998). En el contexto de modelado de Empresas, Mark S. Fox y Michael Gruninger en *Enterprise Modelling* (Fox y Gruninger, 1998) definen EM como una representación computacional de las estructuras, actividades, información, procesos, recursos, personas, comportamiento, metas y restricciones de una empresa de negocios, gubernamental o de cualquier otro tipo. Un EM debe servir de base para comprender su funcionamiento y estructura, que es el punto de partida para introducir mejoras, rediseñar procesos, y en general para lograr el objetivo de toda organización: mantenerse competitiva (Nagel y Dove, 1991). Desde este punto de vista, un EM juega un rol importante, solo en la medida que el mismo permita mostrar todos los aspectos de la misma, consultar sobre el funcionamiento actual de los procesos que la componen, y evaluar posibles cambios en los mismos. Para Curtis (Curtis y otros, 1992) un EM, está formado por la descripción de los procesos de negocio que se llevan a cabo en la empresa, los cuales pueden ser estudiados desde diferentes puntos de vista, y según cual fuera el objetivo del estudio, influenciarán sobre la construcción de los mismos así como también sobre las técnicas usadas para analizarlos. Básicamente un modelo debe ser capaz de proveer información sobre los procesos estudiados, por ejemplo: qué acti-

vidades comprende el proceso, quién ejecuta dichas actividades, cuándo y dónde esas actividades se ejecutan, cómo y por qué son ejecutadas, y qué elementos de datos son manipulados.

Hacia el año 2000, las empresas se encuentran en la necesidad de transición desde un modelo de negocio competitivo a uno colaborativo (Anderson, 2002). Más allá de la necesidad de la reorganización interna, las empresas entienden que el éxito de la compañía es cada vez más dependiente de las colaboraciones y coordinación con sus proveedores así como también con sus clientes, donde la empresa pueda beneficiarse de las oportunidades de integración global, obteniendo como resultado una organización efectiva en costos con oportunidades de ampliar los mercados (Arrazola, 2007). Está claro que aquellos procesos que contribuyen al valor agregado del producto para el cliente siguen siendo un reto para las compañías modernas. Estos procesos comienzan en el proveedor, continúan al agregarle valor estratégico y finalizan cuando el producto final es entregado al cliente. La administración efectiva de estos procesos es crucial en todas las empresas. Hoy día estos procesos muchas veces trascienden los límites de la compañía, e involucran planeamiento e implementación inter-empresa (McCormack y Johnson, 2001).

Así, las empresas cambian sus estructuras con el objetivo de poder alcanzar las demandas cada vez más exigentes e incrementales de los clientes, en este sentido, éstas celebran alianzas con otras dando como resultados estructuras dinámicas y ágiles tales como empresas extendidas (Childe, 1998)(Browne y Zhang, 1999)(Camarinaha-Matos y Afsarmanesh, 1999), empresas virtuales(Camarinaha-Matos y Afsarmanesh, 1999)(Bakos, 2000) (Camarinha-Matos, 2002)(Chalmeta y Grangel, 2003); (Fischer y otros, 2004) (Wu y Su, 2005), empresa globalmente integrada (Arrazola, 2007) y redes de empresas (Mallidi y otros, 1999) (Mezgar y otros, 2000) entre otras estructuras.

De esta manera, en el ámbito de la empresa, se va generando un corrimiento hacia

una perspectiva externa de la misma: no solo es necesario mejorar la organización interna sino también mirar hacia fuera encontrando colaboración con proveedores, clientes y pares.

Ahora bien, todo lo que requiere ser integrado y coordinado necesita ser modelado; en este sentido, el EM es un prerrequisito para lograr la cooperación y la integración. Con el objeto de construir un modelo de empresa que abarque todo el conocimiento de la empresa, la primera cuestión a resolver es que elementos se necesitan para lograrlo. La construcción del EM, es conocido como modelado de empresa y se define como el conjunto de actividades o procesos usados para desarrollar las diferentes partes de un EM (Vernadat, 1996), éste es visto como una disciplina de ingeniería (Petit, 2002) y como tal requiere:

- Metodología de ingeniería de empresa: establecen lineamientos y reglas generales para la representación de un modelo de la empresa orientado a la integración de la misma (Bernus y Nemes, 1997)(CEN, 1996).
- Lenguaje de modelado de empresa: presentan la sintaxis y semántica de los conceptos usados en la construcción de un EM (Vernadat, 2002)(Scheer, 1992).
- Entornos de software para el modelado de empresa: estos son soluciones de software que facilitan a un usuario la tarea de construir el EM. En general presentan entornos gráficos y guías de construcción del modelo, representación de los conceptos del lenguaje adoptado, reuso de componentes para facilitar la definición, etc. como ser por ejemplo cimTool, FirstStep, METIS, MO2GO, e-MAGIM entre otras.

Estos modelos de empresa son la fuente más importantes para derivar los requerimientos de información de una organización. Sin embargo, no solamente se necesita una adecuada representación de la organización en término de procesos, flujo de información,

roles de usuarios, etc, sino que también es importante contar con capacidad de simulación para la interpretación del comportamiento dinámico de la misma (Johannesson, 2007). Simulación es la herramienta que se utiliza para poder analizar los procesos de la empresa, dado que la misma permite estudiarlos tanto en su configuración actual como también bajo un conjunto amplio de alternativas posibles. De esta manera, los resultados de un estudio de simulación pueden ser utilizados por el experto en negocios para seleccionar con mayor precisión y seguridad la mejor forma de conducir la organización. Dentro de las empresas de la nueva generación (conocidas como "2k-enterprise"), la simulación es considerada uno de los factores claves para la supervivencia de la compañía, gracias a su capacidad de "predecir el futuro" (Terzi y Cavalieri, 2004). Simulación es la única técnica que permite tanto un análisis profundo de la situación actual como también una visualización de las posibles alternativas.

La simulación de procesos de la empresa permite:

- generar escenarios, (what if?) lo que implica la elaboración de modelos en los que sea posible analizar los procesos de la empresa en distintas situaciones alternativas como ser por ejemplo, cómo reacciona un proceso ante la ausencia de un recurso dado, el aumento en la demanda o la presencia de un nuevo recurso

- consultar al modelo sobre estimaciones de utilización de recursos en un proceso, demoras producidas en la ejecución de tareas, la utilización de posibles recursos alternativos, etc.

- plantear modificaciones en los procesos (agregar o quitar una tarea, cambiar el orden de ejecución de las tareas, etc.) y evaluar el comportamiento de los mismos.

La ejecución de estos procesos brinda información para determinar la necesidad de:

- incorporar nuevos recursos

- cambiar el orden de ejecución de las tareas,
- asignar diferentes responsabilidades a las personas que intervienen
- incorporar restricciones en la ejecución de una tarea
- automatizar actividades
- combinar tareas duplicadas
- tercerizar actividades ineficientes
- mejorar la performance

La disponibilidad de modelos de procesos ejecutables (es decir que puedan ser simulables) hace posible que las decisiones operativas y estratégicas estén respaldadas por información cuantitativa obtenida de la simulación.

Una variedad de modelos de simulación han sido propuestos en la literatura de sistemas de información tales como redes de petri, aproximación basada en roles (tales como "role activity diagram"), simulación de eventos discretos (DES), etc. Todas estas aproximaciones fueron ampliamente usadas en modelado y simulación de procesos de empresa. Sin embargo, construir un modelo de simulación de los procesos de la empresa, no es una tarea fácil, ya que requiere entender no sólo la organización en su conjunto, esto es, sus procesos, disponibilidad de recursos, las restricciones, las políticas, etc, sino también es necesario tener conocimientos sobre técnicas de simulación. Mucha de la información que se requiere para construir este modelo de simulación (SM) está contenida en el EM, entonces una replicación de ésta en un SM no solo representa una pérdida de tiempo sino también puede producir inconsistencias por la duplicación de la misma. Por otro lado, la estructura de la organización es un factor influyente en la construcción de un modelo de simulación de los procesos de la empresa. Cuando la organización tiene

una estructura distribuida, como por ejemplo una cadena de suministro, o un esquema descentralizado, la complejidad inherente al EM y la cantidad de actores que intervienen dificultan considerablemente el diseño y la construcción del modelo de simulación apropiado para analizar los procesos de la empresa. Uno de los principales problemas que se encuentran al construir un SM para este tipo de empresas es la falta de información proveniente de las partes que conforman la misma. Generalmente las partes intervinientes son reacias a dar a conocer sus datos internos y solo se tiene información superficial del funcionamiento de cada una de estas. Esta falta de información muchas veces lleva a construir SM simples que representan solo un aspecto de la organización como ser por ejemplo sus procesos colaborativos.

En la literatura se ha publicado mucho sobre simulación de cadenas de suministro como un aspecto separado, o no relacionado, con la simulación de procesos de empresa. Así, algunas aproximaciones (Liu y otros, 2004)(Byrne y Heavey, 2004)(VonMevius y Pibernik, 2004)(Biswas y Narahari, 2004)(Cope y otros, 2007) representan el modelo de simulación de cadenas de suministro con modelos locales constituidos por nodos y arcos donde los nodos representan un comportamiento simplificado de las empresas miembros de la cadena y los arcos las relaciones entre estas. En general están orientadas a analizar ciertos aspectos de la cadena como ser: diseño de la red, planificación de la cadena de suministro, planificación del inventario, logística, entre otras. Por otro lado la complejidad del modelo y el tamaño que éste adopte no es menos importante a la hora de ejecutarlo sobre una máquina, es por ello que estos modelos locales necesitan ser simples de manera de poder correr en un computador. En algunos casos se proponen modelos analíticos para la representación de los nodos como se presenta en Lee y otros (2002).

Hacia el año 2000 se reconoce la importancia de usar simulación distribuida como solución para crear modelo complejos que crucen los limites de la empresa, sin necesi-

dad de compartir información (Gan y otros, 2000). La simulación distribuida aparece como la forma de encapsular el conocimiento y comportamiento de cada nodo de la cadena en modelos locales dentro de un gran modelo complejo de simulación (Jian y otros, 2004) (Brun y otros, 2002). Lamentablemente la dificultad de interoperabilidad entre los modelos, y la ausencia de interoperabilidad entre los diferentes paquetes de simulación COTS ("Comercial off the shell") (CSP) condujo a que la mayoría de los trabajos presentes en el campo de simulación de cadenas de suministros estén en etapas de investigación. En su gran mayoría estos trabajos presentan ejemplos de aplicación de simulación distribuida en empresas de manufactura, como la automotriz y semiconductores.

El estándar de la IEEE HLA 1516 (IEEE, 2000b) es ampliamente utilizado para solucionar los problemas de interoperabilidad a nivel sintáctico, sin embargo, las implementaciones particulares de RTI (Run Time Infrastructure) muchas veces son dependientes del proveedor, comprometiendo así la interoperabilidad sintáctica de los simuladores. Esto quiere decir que para un dado RTI, un simulador puede interoperar con otros pero no es posible que este simulador ejecute bajo otro RTI de un proveedor diferente al primero.

Hoy día los estudios de interoperabilidad apuntan a emplear HLA como el estándar de facto, y los trabajos que se están realizando están agrupados en lo que se ha dado en llamar Evolved HLA donde se incorporan conceptos de web services en el estándar HLA 1516. También mucho trabajo se está haciendo para lograr interoperabilidad en COTS. El Simulation Interoperability Standards Organization's (SISO) y el CSP Interoperability Product Development Group (CSPI PDG) han lanzado una primera versión del IRM (Interoperability Reference Model)(Taylor y otros, 2007) el cual provee un marco de referencia para permitir interoperabilidad semántica entre los diferentes paquetes de simulación COTS (CSP) bajo HLA.

En lo relativo al modelo conceptual de Cadenas de suministro, el estándar SCOR (Harmon, 2003), propuesto por el Supply Chain Council (SCC), es empleado para representar el modelo de simulación. Por ejemplo, Fayez y otros (2005), White y Barnett (2003) y Persson y Araldi (2007) emplean este modelo para representar y analizar la cadena incorporando capacidad de simulación. SCOR está orientado a representar procesos estándar que se detectan en las cadenas de suministros como por ejemplo Make, Source, Deliver entre otros. Estos procesos pueden ser vistos como roles que juegan los distintos componentes en una cadena. Terzi (Terzi y Cavalieri, 2004) presenta una importante revisión del uso de simulación en cadenas de suministro, observando el uso de sistemas de simulación de eventos discreto en la mayoría de los trabajos evaluados. Además, se resalta el problema de la necesidad de compartir información para obtener buenos resultados de un estudio de simulación.

En definitiva, de acuerdo a lo expresado en los párrafos anteriores, se observa que el modelado y simulación de cadenas de suministro es tratado como un caso separado del modelado y simulación de los procesos de la empresa, cuando la realidad indica que una cadena de suministro es también una empresa con una estructura particular.

1.2. Objetivos

Frente a la problemática planteada en el apartado anterior, podrían surgir diferentes alternativas de solución, pero en todas ellas, la necesidad de plasmar el conocimiento de la empresa y de poder evaluar sus procesos a través de simulación es un rasgo dominante.

Llamaremos Modelo de empresa ejecutable (E^2M) al modelo compuesto por el modelo de empresa más el modelo de simulación de sus procesos (Gutierrez y Leone, 2006). Una característica importante que debe cumplir este modelo de simulación, es

ser derivado no solo de la vista de procesos del EM sino también de las otras vistas que conforman el EM como ser la vista de recursos. Esto es, un modelo de simulación que contempla los procesos, las relaciones de ellos entre sí, y con los recursos. Si este E^2M es ejecutado en un ambiente distribuido tomará el nombre de modelo de empresa ejecutable y distribuido (DE^2M)(Gutierrez y Leone, 2008). Así un DE^2M es un modelo de simulación de los procesos de la empresa, que puede interoperar con otros softwares de simulación bajo algún protocolo de simulación distribuida.

A partir de estas definiciones se plantea el objetivo principal de esta tesis: *proponer una arquitectura de software que permita la construcción de modelos de empresa ejecutables y distribuidos, teniendo en cuenta la estructura que la empresa puede adoptar.*

En este contexto el primer paso necesario es *seleccionar un lenguaje de modelado de empresa* que sea lo suficientemente simple para facilitar la tarea de desarrollar los modelos, amigable en el sentido de utilizar un lenguaje acorde al conocimiento del usuario, suficientemente robusto para permitir la representación de los distintos aspectos y flexible para permitir adaptar el vocabulario a un dominio de empresa específico. Este lenguaje facilitará la tarea de construir el EM.

En segundo lugar, se requiere *seleccionar las posibles vistas* a ser generadas.

En tercer lugar, se plantea la necesidad de *incorporar capacidad de simulación de los procesos de la empresa* para analizar su comportamiento dinámico. De esta manera, la información que se requiera para construir el modelo de simulación, debe ser extraída de los procesos definidos en el EM.

En cuarto término se plantea el requisito *de construir el modelo de simulación en forma automática* derivado del modelo conceptual, ya que es escaso el conocimiento sobre técnicas de simulación que presenta un experto en negocios.

En quinto lugar, dado que la empresa debe integrar sus procesos internos como también aquellos que trascienden los límites de la organización, se plantea *la necesi-*

dad de poder analizar las características internas de la empresa a través de su E^2M permitiendo la ejecución local de este modelo. Derivado de este objetivo, se plantea la necesidad de *lograr la interoperabilidad de los modelos de simulación* que representan los distintos componentes de la empresa, en un modelo mas abarcativo. Éste será un modelo de simulación distribuida, donde, el comportamiento de los modelos es expuesta a otro software de simulación a través de protocolos de simulación distribuida (Verbraeck, 2004). Estos dos últimos requerimientos pueden formularse en uno: *la necesidad de permitir el análisis de los procesos más internos de la empresa a través de la ejecución de la simulación en un entorno local como también analizar los procesos que trascienden los límites a través de simulación en un ambiente distribuido.*

Por último se quiere indicar la necesidad de *permitir un desarrollo modular* de los modelos dado la gran complejidad de los mismos. Para ello es necesario *definir bloques de construcción* para el EM y *reglas para la generación* del MS.

1.3. Principales Contribuciones

El aporte principal de esta tesis es la definición de la arquitectura para la definición, desarrollo y ejecución de los DE^2M. La arquitectura propuesta, define los componentes necesarios para la construcción modular del DE^2M y su ejecución en un entorno distribuido. Las ventajas de construir un DE^2M pueden ser resumidas en:

- Desarrollo modular del modelo de simulación.
- Permite mantener la independencia y la privacidad de la información en los miembros de la empresa.
- Permite representar todos los aspectos de los diferentes miembros, esto es, sus procesos, recursos, debilidades, políticas, planes, capacidades, etc. Los módulos

se consideran cajas negras donde sólo se expone el comportamiento del mismo a otros módulos.

- No requiere disponer de una alta capacidad de procesamiento dado que la ejecución es distribuida.
- Permite obtener una visión integral del funcionamiento de los procesos de la empresa tanto los internos como los inter-empresa.
- Las interacciones entre los módulos se representan a través de roles, esto permite implementar diferentes roles a un módulo sin necesidad de re-programación del mismo.
- Seleccionar las métricas de interés que se quieren obtener para analizar el modelo.
- Construir el modelo con un lenguaje orientado a negocios sin necesidad de tener conocimiento sobre simulación.
- Obtener una solución rápida y en tiempo del análisis de una empresa, sin necesidad de tener que encarar un proyecto específico para ello.

A continuación se presentan los pasos seguidos en el desarrollo de esta tesis para lograr cada uno de los objetivos planteados en el apartado anterior.

En primer término, se analizó un lenguaje de modelado de empresas desarrollado en este centro de investigación llamado Coordinates (Mannarino y otros, 1999). Coordinates es un lenguaje soportado por el paradigmas de orientación a objetos con capacidad para representar un dominio de empresa en diferentes dimensiones. Presenta las bondades de los lenguajes gráficos, permitiendo definir los elementos del modelo en diferentes niveles de abstracción. Este lenguaje está abierto a las particularidades de cada empresa.

Posteriormente, se determinó que las vistas propuestas en Coordinates, a saber: vista del dominio, vista de tareas y vista dinámica son suficientes para poder representar los aspectos más relevantes de una organización y estudiar sus procesos a través de simulación.

Luego, se combinaron técnicas de modelado de empresa con técnicas de simulación de procesos de empresa. Principalmente se analizaron distintos paradigmas de simulación los cuales serán presentados en el capítulo 2. Al indicar la necesidad de plantear capacidades de simulación se consideraron diferentes clasificaciones de modelos de simulación:

- Modelos discretos, donde las variables de estado cambian instantáneamente en puntos separados en el tiempo.
- Modelos continuos, donde las variables cambian continuamente con respecto al tiempo.

En el caso en que el sistema es una empresa, el cual posee una dinámica particular y gran complejidad, donde todas las actividades se deben a la ocurrencia asíncrona de eventos discretos algunos controlados y otros no (como puede ser por ejemplo la llegada de un cliente o la rotura de un equipo) se está en presencia de un sistema que puede modelarse con una perspectiva discreta. Este enfoque permite analizar la empresa en un nivel de abstracción adecuado para poder manejar la alta complejidad del sistema. En la teoría de sistemas de eventos discretos (DES)(Law y Kelton, 1991)(Zeigler, 1976) se encuentran una serie de metodologías que permiten atacar los problemas de modelado, simulación y análisis de un sistema, tales como Redes de petris (Murata, 1989), Statechart (Harel, 1987), Grafos de eventos (Law y Kelton, 1991), DEVS (Discrete EVents System specification) (Zeigler y otros, 2000b), etc. Se debe tener presente que los EM no solo son complejos por la cantidad de objetos que se manejan (solucionado por la

descomposición jerárquica) sino que su principal dificultad radica en la diversidad de elementos que intervienen en los mismos. Por otro lado, con relación a su comportamiento, estos modelos presentan propiedades de procesos distribuidos y concurrentes. Dadas estas características, la metodología de simulación seleccionada fue DEVS, la cual resulta adecuada para atacar los problemas de complejidad y diversidad y que puede ser fácilmente adaptable a entornos de simulación distribuidas. Consecuentemente se seleccionó una metodología de simulación con las siguientes características:

- Permite una representación modular
- Permite una construcción jerárquica de modelos
- Permite acoplamiento entre modelos
- Permite el reuso de componentes
- Se adapta a simulación distribuida
- Administra eficientemente la ocurrencia de eventos concurrentes permitiendo una ejecución paralela de los mismos.
- Cuenta con un poder expresivo que minimiza los esfuerzos en la construcción de los modelos ejecutables.
- Permite una implementación orientada a objetos.
- Permite desacoplar el modelo de su simulador.

Para lograr la generación del modelo de simulación a partir del modelo conceptual, se definieron bloques de construcción del modelo de simulación que representan comportamientos de los conceptos que intervienen en el EM. Además se propuso un conjunto

de reglas de transformación desde el modelo conceptual de empresa al modelo de simulación. Por otro lado, se definieron métricas de interés para analizar el comportamiento de los procesos de acuerdo a los requerimientos planteados de análisis de los mismos.

A continuación, se utilizaron diferentes máquinas de simulación para entornos locales y distribuidos. Éstas máquinas de simulación son las encargadas de interpretar al modelo de simulación. Así, en un ambiente local, la máquina de simulación local es la que debe interpretar al modelo mientras que en un ambiente distribuido lo hace la máquina de simulación distribuida. Este enfoque permite que los modelos de simulación puedan ser reusados en uno y otro entorno, para el caso del modelo de simulación distribuido, el modelo de simulación local forma parte de un Federado que participa de la simulación distribuida.

Finalmente se plantea un caso de estudio para mostrar las características del DE^{2}M. Para el mismo se considera una cadena de suministro (SC), la cual puede ser vista como una empresa virtual (Villa, 2001): *una red temporaria de compañías independientes vinculadas por tecnología de información para compartir capacidades, costos y acceso a sus respectivos mercados* . Cada miembro de la cadena puede representarse con relación a los otros integrantes, con el iceberg de la figura 1.1.

Se puede observar que la parte visible es de apenas un 10 %, esto corresponde a las relaciones de colaboración entre los miembros de la cadena, pero el 90 % restante correspondiente a flujos de información, materiales y servicios de la cadena que permanecen ocultos en los procesos internos de sus miembros.

Generalmente cuando se analiza una cadena de suministro, sólo se considera ese 10 % visible, es decir sus relaciones con los otros miembros dejando de lado los procesos internos que se llevan a cabo dentro de cada nodo y que constituyen la parte más importante del funcionamiento de la misma.

Figura 1.1: Representación de un nodo de una SC.

El desarrollo de un DE^{2}M para analizar esta cadena procura dar respuesta al problema del iceberg. Cada miembro de la SC utilizando la arquitectura propuesta, define su EM empleando el lenguaje orientado a negocios (Coordinates). Este EM, formado por los distintos modelos (de recursos, de tareas, de estados) representan la estructura interna de cada nodo. Una vez logrado este paso, es posible crear el E^{2}M para analizar el comportamiento interno de los procesos del nodo, evaluándolos frente a posibles cambios en las demandas, en los recursos, en el personal, en las capacidades, etc. En este nivel se está haciendo uso de la capacidad de simulación de los procesos en un ambiente local. A partir de este estudio pueden surgir mejoras de estos procesos internos. Luego, es necesario analizar cómo el comportamiento de estos procesos impactan sobre el comportamiento total de la cadena de suministro. Para ello el E^{2}M es reusado y transformado en un DE^{2}M, esto es, el comportamiento de los procesos se expone a otro software de simulación a través de un protocolo de simulación distribuido. Este modelo necesita definir el conjunto de interacciones que es capaz de "enviar / recibir" "hacia / desde" otros simuladores a través del protocolo de simulación distribuido. El conjunto de interacciones están definidos en los roles que los simuladores locales pueden jugar. En la figura 1.2 se muestra simbólicamente la representación interna del nodo con esta

propuesta. Así la estructura interna del nodo queda completamente representado por el DE^{2}M mientras que la parte visible queda representado por los roles jugados por los integrantes de la cadena.

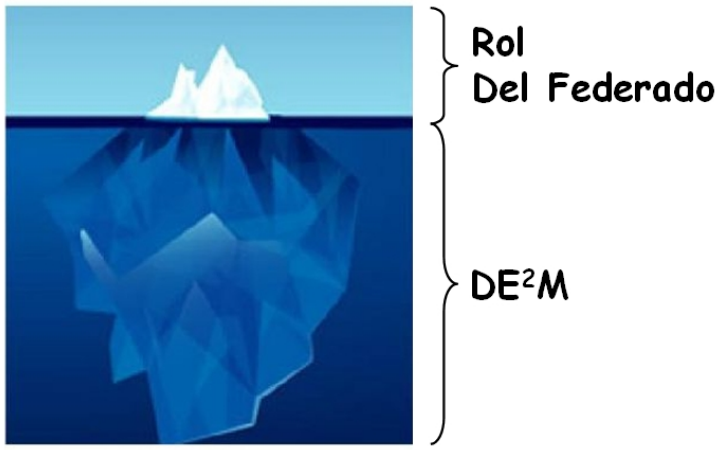


Figura 1.2: Interpretación de un nodo de la SC usando DE^{2}M

En el esquema descripto, cada nodo define su propio modelo, y al interactuar unos con otros dan como resultado el comportamiento de toda la cadena. La figura 1.3 muestra un esquema que representa este concepto.

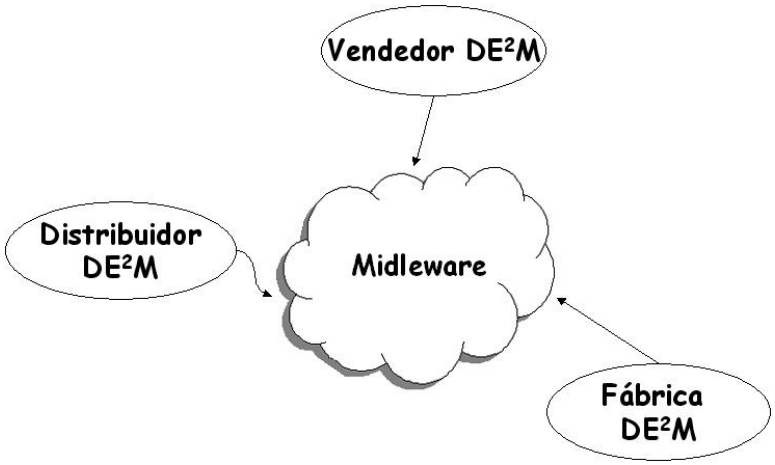


Figura 1.3: DE^{2}M de una cadena de suministro

Los resultados obtenidos y las contribuciones realizadas como parte del trabajo de investigación de esta tesis se divulgaron a través de las siguientes publicaciones:

1. *Coordinates Workbench: A tool for business process modelling.* Gutiérrez, M. Mannarino, G. Leone, H. Anales Jaiio 2000. Pag. 63 a 78.

2. *Coordinates Workbench: an Object-oriented architecture of a tool for conceptualizing an organization* Gutiérrez, M. Mannarino, G. Leone, H. Anales XXI international conference of the Chilean Computer science society. IEEE computer society 2001. ISBN:0-7695-1396-4 pg. 133-142.

3. *Coordinates Workbench: una herramienta para soportar el modelado de Empresa* Gutiérrez, M. Mannarino, G. Leone, H. Anales IDEAS 2001. pg. 299 a 310 ISBN: 9968-32-000

4. *Utilización de Modelos DEVS para simular Procesos de Empresas* Gutiérrez, M. and Leone, H. Anales JIISIC Jornadas Iberoamericanas de Ingeniería del Software e Ingeniería del conocimiento. 2003.

5. *Una Arquitectura para el modelado y análisis de procesos de Empresa.* Gutiérrez, M. Leone, H. Anales JAIIO Jornadas Argentinas de Informática e Investigación Operativa. Simposio Argentino de sistemas de Información. 2004. ISSN:16661141

6. *Supply Chain Process Analysis Using Object-Oriented simulation Tool* Gutiérrez, M. Leone, H. Anales JAIIO Jornadas Argentinas de Informática e Investigación Operativa. Simposio Argentino de tecnología. 2006. ISSN:1850-2776

7. *An Environment for Developing Executable Enterprise Model* Gutiérrez, M. Leone, H. proceeding of the summer simulation conference 2006. ISBN: 1-56555-307-1

8. *Adapting an Executable Enterprise Model Environment for HLA*. Gutiérrez, M. Leone, H. Proceeding of the EMSS European Modelling and Simulation Symposium. 2007. ISBN:88-900732-6-8

9. *E2MDEVS: Modelos de Empresa Ejecutables y distribuidos* Gutiérrez, M. Leone, H. Anales JAIIO Jornadas Argentinas de Informática e Investigación Operativa. Simposio argentino de tecnología. 2007.

10. *DE^{2}M: A Solution for Analyzing Supply Chain* Gutiérrez, M. Leone, H. Proceeding of the WSC Winter Simulation Conference. 2008. ISBN:978-1-4244-2708-6

11. *Using distributed simulation for analyzing Enterprise Models*. Gutiérrez, M. Leone, H. 34 CLEI Conferencia Latinoamericana en Informática. 2008. ISBN: 978-950-9770-02-7

1.4. Organización de la Tesis

El capítulo 2 describe someramente los conceptos de simulación empleados en el desarrollo de la tesis. Se hace una síntesis de los distintos paradigmas de simulación considerados, se realiza una comparación de los mismos, justificándose la elección del formalismo DEVS para el desarrollo del modelo de simulación. Además, se introduce el concepto de simulación distribuida, sus usos, criterios en el modelado temporal y se presenta en detalle el estándar HLA como protocolo de simulación distribuida.

El capítulo 3 presenta la arquitectura propuesta para el desarrollo y ejecución de los modelos DE^{2}M propuestos en esta tesis. Esta arquitectura es una arquitectura en capas que permite ocultar el proceso de simulación al experto en procesos de negocios. Se explicará cada componente de la arquitectura y la metodología de desarrollo empleada para la generación de DE^{2}M con esta arquitectura.

El capítulo 4 describe los conceptos y características utilizadas en la arquitectura para dar soporte al desarrollo del modelo conceptual. Se describe la capa de modelado conceptual en detalle. Esta capa está encargada de la definición y construcción del modelo conceptual de empresa utilizando el lenguaje Coordinates.

El capítulo 5 aborda la descripción de los bloques de construcción del modelo de simulación. Éstos son modelo DEVS con interfaz y comportamiento bien definidos utilizados por la capa de simulación para la construcción del modelo de simulación

El capítulo 6 aborda la transformación del modelo conceptual en un modelo de simulación y se presenta el diseño de la capa de simulación (Simulation Layer) de la arquitectura. Describe las reglas usadas para crear el modelo de simulación a partir de la información contenida en el modelo conceptual de empresa. Se presentarán las máquinas que interpretan los modelos de simulación en los distintos entornos y se abordará la construcción del esquema que interviene en entornos de simulación distribuida. Por último se explicarán los roles definidos y cómo son utilizados para facilitar la interacción entre los diferentes módulos locales.

El capítulo 7 presenta un caso de estudio de una cadena de suministro. Se describe la empresa con sus procesos, recursos y los procesos externos con los otros miembros de la cadena. Se presentan los objetivos de la cadena, políticas y las colaboraciones definidas entre sus miembros para alcanzar los objetivos. Se detalla la construcción del modelo de empresa para uno de esos miembros y se realiza la simulación local. Del análisis de resultados surgen posibles mejoras a los procesos los cuales son implementados y nuevos resultados obtenidos. Luego se muestra una simulación distribuida y se analizan los resultados.

Por último el capítulo 8 presenta las conclusiones de la tesis y los trabajos futuros.

2

Paradigmas de Modelado y Simulación de Eventos Discretos

En este capítulo se hace una revisión de los distintos paradigmas de simulación más comúnmente usados en la simulación de eventos discretos. Se realiza una comparación entre ellos y se justifica la utilización de DEVS como formalismo empleado para desarrollar esta propuesta. Se hace una introducción al concepto de simulación paralela y distribuida, sus usos, los problemas que aparecen y las diferentes propuestas que surgieron para su solución. Se introduce el estándar HLA y su uso en simulación distribuida.

Los paradigmas de modelado y simulación de eventos discretos proveen una manera de representar problemas centrando el foco de interés en el comportamiento dinámico de los sistemas. Cuando la dinámica es compleja, los sistemas se representan usando eventos discretos en el tiempo, donde un evento representa un cambio instantáneo en el sistema. Modelar un sistema de eventos discretos, consiste en representar el comportamiento del sistema a través de una estructura sintáctica. Los formalismos existentes proponen diferentes sintaxis de representación con una semántica asociada que describe el significado. Dichos formalismos tienen características similares, por ejemplo, todos ellos en la descripción de un modelo de simulación, usan conceptos tales como conjunto de estados, eventos de entrada y de salida, transiciones de estado que son generadas por

la ocurrencia de eventos, y la noción del paso del tiempo. La simulación es un proceso que intenta predecir el comportamiento de un sistema creando un modelo aproximado de este.

La simulación es un proceso que tiene tres etapas: modelado, ejecución y evaluación. La primer etapa se realiza utilizando algún lenguaje de modelado que permita representar las características más sobresalientes del sistema que se desea evaluar. La segunda etapa está asociada al programa de computadora usado para representar el modelo de una manera que pueda ser interpretado por la computadora. En esta etapa intervienen conceptos como eventos, estados, transiciones de estados, métricas, número aleatorios, objetos, etc. La última etapa, la de evaluación, se realiza observando e interpretando los resultados obtenidos de la ejecución de un modelo de simulación. La validez de los resultados dependerá de la validez del modelo desarrollado como también de los datos que alimentaron al sistema en la etapa de simulación.

La simulación de eventos discretos, considera que el modelo de simulación está compuesto por un conjunto de *eventos* y *variables* que representan sus estados. Los eventos ocurren en un determinado tiempo indicado por su *marca temporal*, durante el tiempo de simulación, y son organizados en la *lista de eventos* de acuerdo a esta. El proceso de simulación consiste en tomar el primer evento de la lista de eventos, avanzar el tiempo de simulación al tiempo establecido en la marca temporal del evento que se está tratando y finalmente ejecutar el evento lo que producirá un cambio de estado en el modelo de simulación.

El proceso de simulación que sigue este esquema, garantiza el tratamiento de los eventos en un orden cronológico adecuado, esto es, se respeta el orden en que los eventos ocurren en el sistema, dado que existe una única lista de eventos ordenado. Luego si el evento e con marca temporal t es ejecutado, esto quiere decir que no existen eventos con marca temporal menores a t en la lista de eventos, entonces al ejecutarse el eventos

e, el tiempo de simulación avanza a t. Si como consecuencia de este paso, se generan nuevos eventos, los mismos tendrán una marca temporal igual o mayor a t, estos eventos son incorporados a la lista y se la ordena nuevamente repitiéndose el proceso.

2.1. Paradigmas de Simulación

En este punto se tratarán las diferentes metodologías propuestas en la bibliografía consultada para la simulación de procesos complejos.

2.1.1. Redes de Petri

Las redes de Petri (Petri, 1962) fueron definidas en 1962 por Carl Adams Petri como una herramienta para modelar y analizar procesos. Una de las cualidades que hace fuerte a estas redes es permitir la descripción de los procesos de una manera gráfica, sin descartar la sólida base matemática que la sustenta. A diferencia de otras técnicas, están enteramente formalizadas. Gracias a esta formalización, es posible expresar las propiedades del proceso que se modela de una manera robusta. En los años subsiguientes a la presentación del formalismo, el modelo propuesto por su autor, ha sido expandido en diferentes formas, gracias a lo cual es posible modelar procesos más complejos en una manera accesible, estas extensiones se conocen como Redes de Petri de nivel superior.

2.1.1.1. Redes de Petri Clásicas

Se le da el nombre de Redes de Petri Clásicas al formalismo original para distinguirlas de las derivaciones de las mismas que surgieron a posteriori.

Una red de Petri consiste de lugares (indicados con círculos), transiciones (indicados con rectángulos), arcos (indicados con líneas dirigidas) y tokens (representados por puntos negros). Los lugares representan estados, lugares físicos, medio, buffer, fase o

condición del proceso modelado; son pasivos en el sentido que no pueden cambiar el estado de la red, mientras que las transiciones representan acciones, transformaciones, eventos u operaciones que se ejecutan y que cambian el estado de la red. Un arco conecta un lugar con una transición. Existen dos tipos de arcos: (i) aquellos que tienen como origen un lugar y como destino una transición y (ii) aquellos que tienen como origen una transición y como destino un lugar. No se permiten arcos que vayan de un lugar a otro ni de una transición a otra. Los tokens representan objetos que pueden ser tanto físicos como información. Basado sobre los arcos, se puede determinar el lugar de entrada de una transición. Un lugar P es un lugar de entrada a una transición T si existe un arco dirigido que va de P a T. Similarmente se puede determinar el lugar de salida de una transición. Un lugar P es un lugar de salida para la transición T si existe un arco dirigido que va de T a P. Una transición puede tener más de un lugar de entrada y más de un lugar de salida. Los lugares pueden contener tokens, los cuales son indicados como puntos negros en el círculo que representa el lugar al cual pertenecen. Estos tokens pueden moverse en la red aún cuando la estructura de la red es fija. Cuando una transición se ejecuta, toma tokens de sus lugares de entrada y los mueve a sus lugares de salida. Esta acción es llamada disparo de la transición. Ahora bien, el disparo de una transición está sujeto a reglas que para poder describirlas es necesario introducir el concepto de estado de la red. El estado de una red de Petri está determinado por la distribución de los tokens entre los lugares. Una transición puede ser disparada si la misma está activa, esto ocurre cuando hay al menos un token en cada uno de los lugares de entrada a esa transición, se dice entonces que la transición está lista para disparar. Una vez que la transición se dispara, se elimina un token de cada lugar de entrada y un token es agregado a cada lugar de salida. En otras palabras, disparar una transición consume token en sus lugares de entrada y produce token en sus lugares de salida, generando un nuevo estado de la red. A partir de este nuevo estado se

vuelven a evaluar cuales son las transiciones activas, en caso de que exista más de una transición activa, no es posible determinar cual es la transición que se disparará primero, esta decisión deberá tomarse en función de los propósitos o beneficios que se quieran lograr. Una vez que una transición fue seleccionada y disparada puede suceder que otras transiciones que antes estaban activas dejen de estarlo y que nuevas transiciones se agreguen a la lista de las transiciones activas. Es posible representar con estas redes procesos repetitivos. Un ejemplo de esto se muestra en la 2.1 donde se representa el comportamiento de un semáforo usando este formalismo.

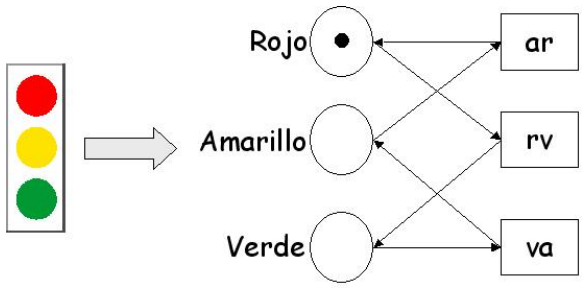


Figura 2.1: Representación del funcionamiento de un semáforo usando una Red de Petri

Los tres posibles estados de un semáforo son representados por tres lugares: rojo, amarillo y verde. Los posibles cambios de luces se representan con tres transiciones: ar (transición de amarillo a rojo), rv (transición de rojo a verde), y vr (transición verde a rojo).

A medida que se complica el proceso a modelar, los lugares y las transiciones crecen demasiado, haciéndolas inaccesibles y en muchos casos imposible de ser utilizadas para modelar una actividad particular. Esto explica el porqué de las expansiones que sufrieron las redes de Petri clásicas en diferentes formas. Las tres extensiones más importantes son:

- extensión del color,
- extensión del tiempo y

- extensión jerárquica.

2.1.1.2. Redes de Petri de Nivel Superior

A) **Redes de petri coloreadas**

Las Redes de Petri coloreadas, es un lenguaje gráfico para diseñar, especificar, simular y verificar sistemas. Es particularmente adecuado para sistemas en los cuales es importante la comunicación, sincronización y el compartir recursos. En estas redes los tokens son utilizados para representar una amplia gama de cosas con la particularidad de que dados dos tokens cualesquiera de estos pueden distinguirse uno de otro usando *colores*.

Así, en una red de Petri clásica, si estuviera modelando un proceso en una empresa de seguros, un token puede representar reclamo de seguro pero entre dos reclamos no es posible hacer distinciones. Una característica de las redes coloreadas es poder asignar al token más información, como por ejemplo, el nombre del cliente, el número de póliza, el valor reclamado, etc. Las redes de Petri coloreadas, obtienen su nombre debido a que en ellas se permite el uso de tokens que contienen datos de diferentes tipos (colores) y que pueden distinguirse unos de otros, en contraste a las redes de Petri clásicas, donde los tokens no pueden distinguirse unos de otros. En el comienzo solo se usaron un conjunto fijo de tipos, más tardes estos tipos fueron creciendo en complejidad y hoy día puede emplearse cualquier valor de dato complejo (por ejemplo listas de cientos de registros, que contengan campos de diferentes tipos de datos.) Cuando una transición se dispara, los valores de los tokens producidos dependerán de los valores de aquellos que fueron consumidos. A diferencia de las redes de Petri clásicas, el número de tokens producidos es variable y estará determinado por el valor de aquellos que son consumidos.

Otra diferencia con las redes de Petri clásicas es que un lugar de salida puede no recibir ningún token cuando su transición asociada se dispara. La decisión de qué tokens son generados y hacia que lugar de salida son enviados, es tomada en la transición. En estas redes se pueden establecer condiciones sobre los valores de los token en los lugares de entrada para que la transición se dispare, luego, una transición va a estar activa, si en el lugar de entrada a la transición existen suficientes tokens y los valores de estos aparean las precondiciones definidas en la transición. Estas precondiciones son expresiones lógicas que relacionan valores de los tokens y que se establecen en el arco que une el lugar de entrada con la transición. Las precondiciones pueden ser útiles para sincronizar las transiciones.

Dependiendo el objetivo para el cual se utilicen las redes de Petri, las precondiciones pueden expresarse en forma de texto, pseudo-código, código en un lenguaje particular, etc.

B) **Extensión del tiempo**

La extensión del tiempo a las redes de Petri clásicas, estuvo inspirada en la necesidad de manejar tiempos en los procesos modelados. Por ejemplo si se está modelando un proceso de producción de autos en una fábrica automotriz, se podría querer conocer el tiempo necesario para completar un auto. Con la extensión del color este tipo de consultas resulta difícil de manejar, por lo que se propuso esta nueva extensión.

En este tipo de redes, los tokens tendrán asociado una marca temporal (timestamp) indicando el tiempo a partir del cual el token estará disponible. Luego, una transición estará activa en un momento dado, si en los lugares de entrada a la misma existen token con una marca de tiempo menor o igual al tiempo actual de

simulación. Los tokens son consumidos sobre una base FIFO (First-In First-Out) con lo cual el token con la marca más temprana es el primero consumido. Al igual que en las redes clásicas, si dos o más transiciones están activas al mismo tiempo, no se indica cuál es la que dispara primero (no se tiene en cuenta el disparo en paralelo de las transiciones.) Cuando una transición se dispara, los tokens producidos tendrán una marca temporal igual o superior al momento en que se produce el disparo, esta demora será determinada por la transición que se dispara, y la misma será dependiente de los valores de los tokens consumidos. Es posible que esta demora se establezca fija o bien al azar. Se considera que el disparo es instantáneo y no consume tiempo. Si bien esta extensión es un importante aporte, está acotado a un dominio, siendo difícil de representar otros aspectos del proceso a modelar.

C) **Extensión jerárquica**

Con las extensiones del color y el tiempo es posible describir procesos complejos, pero la red obtenida no permite visualizar el proceso en sí, es decir se pierde proyección sobre lo que se está modelando, más aún cuando se trata de procesos altamente complejos. Esto se debe a que la red obtenida será muy extensa y no representa la estructura jerárquica del proceso modelado. No hay una manera de representar jerarquía en este tipo de red. La extensión jerárquica propone agregar estructura a la red.

Se introduce el concepto de *Proceso* el cual está representado en el diagrama como un cuadrado con doble línea. Éste representa una subred encapsulando lugares, transiciones, arcos y subprocesos. Debido a que un proceso puede ser construido desde subprocesos, los cuales pueden a su vez ser construidos a partir de otros

subprocesos, es posible estructurar un proceso complejo jerárquicamente. De esta manera un procesos puede adoptar dos formas en la representación:

- como un cuadrado con doble línea formando parte de la definición de un proceso de nivel superior en la jerarquía o
- como la definición del proceso en si mismo (un conjunto de elementos desde los cuales el proceso es definido).

Con esta representación jerárquica, un proceso puede definirse tanto button-up como top-down.

2.1.2. Formalismo DEVS

El formalismo DEVS (Discrete EVents System specification) fue propuesto por Zeigler (Zeigler, 1976), para especificar sistemas de eventos discretos. DEVS, exhibe en forma particular, los conceptos de teoría de sistemas y modelado. Este formalismo ha ganado mucha importancia no solamente por afrontar el problema de modelos de eventos discretos sino también por proveer una base computacional que permite implementar comportamiento expresados en otros sistemas bases como ser tiempo discreto y ecuaciones diferenciales entre otros (Zeigler y otros, 2000b).

El formalismo DEVS con el cual se comienza, es llamado formalismo DEVS clásico para distinguirlo de una revisión que fue introducida después de 15 años llamada DEVS paralelo la cual remueve algunas restricciones que aparecían en el formalismo original que reflejaba la forma de operar secuencial de los ordenadores de aquella época, explotando el paralelismo, elemento crucial en los ordenadores modernos. Esta propuesta brinda un enfoque abierto y flexible basado en conceptos generales sobre dinámica de sistemas, permitiendo la descomposición de modelos con un enfoque jerárquico y modular.

En DEVS se deben especificar: (i) modelos básicos desde los cuales se construirán modelos más complejos, y (ii) conexiones que simbolizan cómo los modelos se conectan unos con otros para definir el comportamiento de un sistema de nivel superior. Esta especificación modular de modelos de eventos discretos propuesta por dicho formalismo requiere que se adopte una visión diferente de la empleada en los lenguajes de simulación tradicionales. Como en toda especificación modular general, los modelos poseen puertos de entrada y de salida a través de los cuales interactúan con el entorno. En el caso de eventos discretos, los eventos son los valores que aparecen en tales puertos. Más específicamente, cuando ocurren eventos externos (fuera del modelo), estos son recibidos en sus puertos de entradas, y la especificación del modelo debe definir cómo va a responder a ellos. También ocurrirán eventos internos (dentro del modelo), que causan cambios de estado, y se podrá generar eventos en sus puertos de salida para ser transmitidos a otros modelos componentes.

DEVS define dos tipos de modelos: Modelos Básicos y Modelos Acoplados. Estos últimos están compuestos por modelos básicos y acoplados satisfaciendo la propiedad de ser cerrados bajo acoplamiento. Esta propiedad establece que la acción de acoplar dos modelos DEVS genera como resultado otro modelo DEVS, permitiendo la construcción jerárquica de sistemas complejos.

El formalismo DEVS paralelo se diferencia de DEVS clásico en cuatro puntos principales:

- Los puertos se representan explícitamente
- Pueden manejar bags de entradas y salidas
- Se agrega una función, llamada confluencia que decide cual es el próximo estado en caso de colisión entre eventos externos e internos.

- Se elimina la función *Select* que impide la ejecución en paralelo de eventos simultáneos.

A continuación se presentan las descripciones de modelos básicos y acoplados en DEVS clásico y en DEVS paralelo.

2.1.2.1. DEVS clásico

Se describen los modelos DEVS básicos y acoplados de acuerdo al formalismo propuesto por Zeigler (Zeigler, 1976). Estos modelos están representados por tuplas de elementos componentes, que definen entradas, salidas y reacciones del sistema ante dichas entradas. Existen dos formas de representar los modelos DEVS clásicos, indicando explícitamente los puertos o sin representar los mismos. En este punto sólo se incluirá la descripción del modelo DEVS clásico con puertos.

Modelo DEVS básico. Un modelo DEVS básico queda definido por una tupla de elementos:

$$DEVS =< X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta > \tag{2.1}$$

Donde:

$X = \{(p, v)\}$; p ϵ InPort y v ϵ Xp, siendo InPort el conjunto de puertos de entrada y Xp los valores de entrada permitidos para el puerto "p".

$Y = \{(p, v)\}$; p ϵ OutPort y v ϵ Yp, siendo OutPort el conjunto de puertos de salida y Yp los valores de salida para el puerto "p".

S: conjunto de estados;

$\delta_{int} : S \rightarrow S$: función de transición de estado interna.

$\delta_{ext} : Q$ x $X \rightarrow S$: función de transición de estado externa.

$\lambda : S \rightarrow Y$: función de salida que genera eventos externos en la salida. Una restricción

impuesta por DEVS es que los eventos de salida solo pueden ocurrir inmediatamente antes de una transición interna

$ta : S \rightarrow \mathbb{R}_0^+ \cup \infty$: función de avance del tiempo

$Q = \{(s, e) | s \epsilon S, 0 \leq e \leq ta(s)\}$ y e es el tiempo que transcurrió desde la última transición de estados.

Modelo DEVS acoplado. Un modelo DEVS acoplado queda definido como una tupla de elementos:

$$Coupled =< X, Y, D, \{M_i \, con \, i \, \epsilon \, D\}, EIC, EOC, IC, Select > \tag{2.2}$$

Donde:

$X = \{(p, v)\}$; p ϵ InPort y v ϵ Xp, siendo InPort el conjunto de puertos de entrada y Xp los valores de entrada permitidos para el puerto "p".

$Y = \{(p, v)\}$; p ϵ OutPort y v ϵ Yp, siendo OutPort el conjunto de puertos de salida y Yp los valores de salida para el puerto "p".

D: conjunto de nombres de componentes. Para cada $i \, \epsilon$ D:

$M_i = \{X_i, Y_i, S_i, \delta_{int_i}, \delta_{ext_i}, \lambda_i, ta_i\}$ es un modelo DEVS

$EIC \subseteq \{((M, ipN)(d, ipd))\} \, con \, ipN \, \epsilon \, InPort, \; d \, \epsilon \, D \, y \, ipd \, \epsilon \, InPort_d$

$EOC \subseteq \{((d, opd)(M, opM))\} \, con \, opM \, \epsilon \, outPort, \; d \, \epsilon \, D \, y \, opd \, \epsilon \, OutPort_d$

$IC \subseteq \{((a, opa)(b, ipb))\} \quad con \, a, b \, \epsilon \, D, \; opa \, \epsilon \, outPort_a, \quad y \quad ipb \, \epsilon \, InPort_b \, y \, a \neq b$

Select: $2^D - \{\} \rightarrow D$ (esta función evita la ejecución en paralelo de eventos concurrentes.)

2.1.2.2. DEVS paralelo

DEVS paralelo es una modificación realizada al formalismo DEVS, que se debió principalmente a la necesidad de representar eventos concurrentes presentes en los sistemas

modernos. Se puede observar que las modificaciones introducidas son casi insignificantes, sin embargo, el impacto que estas tienen sobre el comportamiento de los modelos es muy grande. Los modelos básicos permiten manejar en los puertos de entrada y salida conjuntos de eventos. Esto es, un modelo puede recibir más de un evento en uno de sus puertos de entrada, siendo todos ellos procesados e interpretado por el modelo. Así también un modelo puede enviar más de un evento de salida por sus puertos de salida. Los modelos acoplados solo se diferencian del formalismo clásico por la falta de la función *select*. A continuación se describen cada uno de estos modelos y se hace una breve explicación del funcionamiento del modelo DEVS básico.

Modelo DEVS básico. Un modelo DEVS básico queda definido como una tupla de elementos:

$$DEVS =< X, Y, S, \delta_{int}, \delta_{ext}, \lambda, ta > \tag{2.3}$$

Donde:

$X = \{(p, v)\}$ p ϵ InPort y v ϵ Xp, siendo InPort el conjunto de puertos de entrada y Xp los valores de entrada permitidos para el puerto "p".

$Y = \{(p, v)\}$; p ϵ OutPort y v ϵ Yp, siendo OutPort el conjunto de puertos de salida y Yp los valores de salida para el puerto "p".

S: conjunto de estados;

$\delta_{int} : S \rightarrow S$: función de transición de estado interna.

δ_{ext} : Q x $X^b \rightarrow$ S: función de transición de estado externa.

δ_{con} : S x $X^b \rightarrow$ S: función de transición de confluencia.

$\lambda : S \rightarrow Y^b$: función de salida que genera eventos externos en la salida. Una restricción impuesta por DEVS es que los eventos de salida solo pueden ocurrir inmediatamente antes de una transición interna.

$ta : S \rightarrow \mathbb{R}_0^+ \cup \infty$: función de avance del tiempo

Donde:

$Q = \{(s, e)|\ s\ \epsilon\ S, \quad 0 \leq e \leq ta(s)\}$ y e es el tiempo que transcurrió desde la última transición de estados.

X^b representa múltiples eventos de entrada

Y^b representa múltiples eventos de salida.

La interpretación de estos elementos es la siguiente: en algún instante de tiempo el sistema llega al estado s. Si no ocurre ningún evento externo el sistema permanecerá en dicho estado s durante un tiempo ta(s). Note que ta(s) puede ser un número real, pero también puede tomar los valores especiales 0 e ∞. En el primer caso el tiempo es tan corto que ningún evento externo puede intervenir - se dice que s es un estado transitorio. En el segundo caso, el sistema permanece en el estado s por siempre, a menos que un evento externo interrumpa su inactividad - se dice que s es un estado pasivo. Cuando el tiempo transcurrido, e, iguala a ta(s), el sistema produce un evento de salida con valor λ(s) y cambia al estado $s' = \delta_{int}(s)$. Se dice entonces que el sistema efectuó una transición interna.

Si por el contrario por un puerto de entrada el sistema recibe un evento $x\ \epsilon\ X$ en el tiempo e donde $e <$ ta(s) el sistema cambiará de estado, siendo el nuevo estado s'= $\delta_{ext}(s, e, x)$, en este caso no se genera ningún evento de salida, y se dice que el sistema efectuó una transición externa. De esta manera, la función de transición interna establece el nuevo estado del sistema si ningún evento ocurre desde la última transición, la función de transición externa, establece el nuevo estado del sistema cuando ocurren eventos externos (este nuevo estado está determinado por el evento x que se recibe, el estado s en que se encuentra el sistema y el tiempo e transcurrido desde el último evento.) Luego, el sistema estará en un nuevo estado s' con algún tiempo restante ta(s'). Este ciclo se repite hasta que no existan eventos o hasta que se alcanza una cantidad de ciclos establecida.

Modelo DEVS acoplado Un modelo DEVS acoplado queda definido como una tupla de elementos:

$$Coupled =< X, Y, D, \{M_i \; con \; i \; \epsilon \; D\}, EIC, EOC, IC > \tag{2.4}$$

Donde:

$X = \{(p, v)\}; p \epsilon InPort \quad y \quad v \epsilon Xp$, siendo InPort el conjunto de puertos de entrada y Xp los valores de entrada permitidos para el puerto "p".

$Y = \{(p, v)\}; \; p \, \epsilon \, OutPort \quad y \quad v \, \epsilon \, Yp$, siendo OutPort el conjunto de puertos de salida y Yp los valores de salida para el puerto "p".

D: conjunto de nombres de componentes.

Para cada $i \, \epsilon$ D:

$M_i = \{X_i, Y_i, S, \delta_{int_i}, \delta_{ext_i}, \lambda_i, ta_i\}$ es un modelo DEVS.

$$EIC \; \subseteq \; \{((M, ipN)(d, ipd))\} \tag{2.5}$$

donde: $ipN \, \epsilon \, InPort, \quad d \, \epsilon \, D$ y $ipd \, \epsilon \, InPort_d$

$$EOC \subseteq \{((d, opd)(M, opM))\} \tag{2.6}$$

donde: $opM \, \epsilon \, outPort, \quad d \, \epsilon \, D$ y $opd \, \epsilon \, OutPort_d$

$$IC \subseteq \{((a, opa)(b, ipb))\} \tag{2.7}$$

donde: $a, b \, \epsilon \, D, \quad opa \, \epsilon \, outPort_a$, y $ipb \, \epsilon \, InPort_b$, y $a \neq b$

En un modelo acoplado se debe determinar cuales son los modelos componentes (D) y de que manera éstos están conectados entre sí y con el modelo padre. Estas conexiones están representadas por los tres tipos de acoplamientos: (i) **EIC**(expresión 2.5): acoplamiento externo de entrada (External Input Coupling) determina que puertos de entrada del modelo padre están conectados con puertos de entrada de modelos componentes;

(ii) **EOC**(expresión 2.6): acoplamiento externo de salida (External Output Coupling) determina que puertos de salida de los componentes están conectados con puertos de salida del modelo padre; (iii) **IC**(expresión 2.7): acoplamiento interno (Internal Coupling) describe que puertos de salida de modelos componentes, están conectados con puertos de entrada de otros modelos componentes, en este último caso no se permiten feedback directo, es decir que para un componente d, ninguno de sus puertos de salida puede estar conectado con alguno de sus puertos de entrada. Estos acoplamientos describen las interacciones entre un conjunto de componentes.

La definición de modelos acoplados DEVS paralelo, se especifica de la misma manera que en DEVS clásico, con la única diferencia que se omite la función *select*. Esto es un cambio mínimo, sin embargo su semántica es muy fuerte, ya que difiere significativamente en como se manejan los componentes inminentes: en DEVS paralelo no hay un orden cronológico de estos componentes, todos los componentes inminentes, generan sus salidas las cuales son luego distribuidas usando la información de acoplamiento.

Ciclo de simulación DEVS. DEVS propone separar los modelos de los simuladores de los mismos, de esta manera, cada modelo atómico tiene asociado un simulador y cada modelo acoplado tiene asociado un coordinador. Un Coordinador especial llamado Coordinador Raíz, dirige la simulación. El mecanismo de simulación DEVS consiste de 4 etapas: (i) avanzar la función de tiempo hasta el tiempo del nuevo evento, (ii) calcular las entradas y salidas, (iii) enviar los mensajes a sus respectivos modelos de acuerdo al acoplamiento definido, (iv) ejecutar las funciones de transición interna y externas para cada componente. La figura 2.2 muestra el ciclo de simulación DEVS implementado en el coordinador raíz.

En esta figura se observa que en el primer paso (calcular TN), el coordinador solicita a cada uno de sus componentes que calculen el tiempo de su próximo evento (tN).

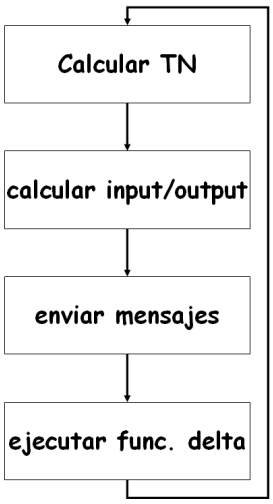


Figura 2.2: Ciclo de simulación DEVS

Luego el tiempo más pequeño es llamado TN* y es el tiempo al que será avanzado el tiempo de simulación. TN* queda representado como:

$$TN^* = min\{tN_d | d \epsilon D\} \tag{2.8}$$

Donde: D es el conjunto de componentes y tN_d es el tiempo del próximo evento del componente d.

A partir del valor de TN*, el coordinador agrupa en el conjunto I a todos los componentes cuyo tiempo del próximo evento es igual a TN*. Este conjunto I es conocido como el conjunto de los componentes inminentes y está representado por:

$$I = \{d \, \epsilon \, D / tN_d = TN^*\} \tag{2.9}$$

En el paso dos (Calcular input/output), el coordinador solicita calcular las salidas a los componentes inminentes. De esta manera, cada uno de los componentes del conjunto I ejecuta su función de salida generando los mensajes que serán entradas de otros componentes.

$$\forall d \, \epsilon \, I \Rightarrow send(computeInputOutput, d)$$

Cuando el coordinador recibe las salidas, usando información de acoplamiento, envía los mensajes a los correspondientes puertos de entrada. En este paso (enviar mensajes), el coordinador genera un nuevo conjunto M de todos los componentes que tienen mensajes en alguno de sus puertos de entrada.

$$M = \{d \,\epsilon\, D \,/\, messageOnPort(p) \neq \emptyset\} \quad \wedge \quad p \,\epsilon\, Inport_d \tag{2.10}$$

Finalmente, el último paso (ejecutar func. Delta) los componentes reciben la orden de ejecutar sus funciones de transición de estado de la siguiente manera: todos los componentes que pertenecen a la intersección de los conjuntos I y M ejecutan la función de transición de confluencia (confluence transition function); todos los componentes restantes en I, ejecutan su función de transición interna y por último todos los componentes restantes en M, ejecutan su función de transición externa:

$$\forall\, d \,\epsilon\, I \cap M \Rightarrow send(\delta_{con}, d)$$

$$\forall\, d \,\epsilon\, M - (I \cap M) \Rightarrow send(\delta_{ext}, d)$$

$$\forall\, d \,\epsilon\, I - (I \cap M) \Rightarrow send(\delta_{int}, d)$$

Este ciclo se repite hasta que no existan componentes inminentes o hasta que un número dado de iteraciones se completen.

Se dispone de implementaciones de DEVS orientadas a objetos tales como el framework DEVSJAVA (ACIMS, 2004) y DEVS C++ (Zeigler y otros, 1996). DEVS fue extendido para ser aplicado en diferentes dominios, dando origen a formalismos tales como: Dynamic Structure DEVS (Barros, 1996) que permite a un modelo cambiar su estructura dinámicamente durante la ejecución; Symbolic DEVS (Chi, 1997) que representa la base del tiempo en una manera simbólica como ser intervalos de tiempos de

arribos o tiempo de servicio; Real Time DEVS (Zeigler y otros, 2000a) que permite que un modelo DEVS sea desarrollado en un entorno de simulación y ejecutado en tiempo real mas que en tiempo de modelado y por último Fuzzy DEVS (Zeigler y otros, 2000a) que provee una versión Fuzzy del concepto de DEVS facilitando una incorporación más sencilla de incertidumbre en el modelo.

2.1.3. Statecharts

Los statecharts (Harel, 1987) fueron concebidos por Harell en 1983 como un formalismo visual para especificar comportamiento reactivo.

Los statecharts extienden los diagramas de transición de estado convencionales con tres elementos esenciales: jerarquía, concurrencia y comunicación, transformando al lenguaje de diagramas de estado en un lenguaje altamente estructurado pero a la vez simple. Estos diagramas son ampliamente empleados para representar el comportamiento de sistemas de eventos discretos. El paradigma de análisis estructurado usa estos diagramas para diseñar el comportamiento de sistemas, mientras que el paradigma de diseño Orientado a Objeto los usa para representar el comportamiento de los objetos (Harel y Gery, 1996). Statecharts tiene como meta construir modelos que sean intuitivos y bien estructurados pero a la vez ejecutables y analizables. Cabe destacar que este formalismo fue adoptado por UML como el medio para especificar comportamiento. El formalismo visual describe estados y transiciones de una manera que permite el agrupamiento, la ortogonalidad (como ser concurrencia), y el refinamiento, dando soporte también a la capacidad de moverse fácilmente entre los distintos niveles de abstracción.

Statecharts utiliza los rectángulos para denotar estados en cualquier nivel, y un rectángulo dentro de otro identifica relaciones jerárquicas. Los arcos dirigidos indican transiciones de estados, éstas pueden empezar y terminar en estados de cualquier nivel.

Estos gráficos están basados en un concepto general, el "higraph", el cual combina nociones de círculos de Euler, diagramas de Venn e hipergráficos. Los arcos dirigidos pueden estar rotuladas con un evento (o abreviación de éste), y opcionalmente con una condición (guard condition) indicada entre corchetes.

Se pueden distinguir dos tipos de estados: los superestados y los estados ortogonales. La semántica de un superestado es la del or-exclusivo, representando un grupo de estados. Cuando se establece que un sistema tiene como estado actual un superestado, el sistema puede estar en sólo uno de los estados del grupo. Este superestado captura comportamiento común a sus estados componentes.

Para determinar cual de los estados componentes del superestado va a ser el actual, existen dos formas: (i) usando el concepto de historia del sistema (conocida como entrada-por-historia) o (ii) definiendo un estado default. La *entrada-por-historia*, significa entrar al estado más recientemente visitado. Esta forma de determinar el estado se indica incluyendo una H en el rectángulo que representa al superestado. Este símbolo indica que solo se aplica al nivel donde fue especificado pero no en los subniveles. Si se quiere que esta estrategia se aplique en un nivel y todos sus subniveles entonces se debe agregar un asterisco junto a la H.

La segundo forma de determinar el estado actual es usar un estado por default como el estado de inicio. Para hacerlo se usa la misma notación que para indicar el estado de inicio en autómatas de estado finitos (circulo negro relleno).

Es posible también introducir componentes AND, capturando las propiedades de sistemas compuestos por subsistemas, donde el hecho de estar en un estado dado implica estar en todos los estados componentes. La notación empleada en statechart para indicar estados ortogonales, es una línea punteada que separa el rectángulo (estado) en componentes. Cada una de estas partes pueden a su vez describir el comportamiento de los subsistemas usando statechart. Esta ortogonalidad, muestra por un lado la sin-

cronización que puede existir entre los componentes de un sistema, haciendo que todos cambien de estado ante la ocurrencia de un evento. Pero también muestra la independencia de cada uno de estos subsistemas cuando cada uno resuelve como responder a un evento independientemente del resto de los subsistemas.

2.1.4. Comparación de los Paradigmas de Simulación

Si bien los statechart basados en la teoría de autómatas de estado finitos, son fáciles de usar, ya que se trata de representaciones gráficas sencillas, cuando se trata de modelar sistemas complejos el número de estados crece considerablemente, haciéndose difícil la lectura y construcción de los diagramas. Aún cuando considera la representación de jerarquías, permitiendo abstraer conceptos genéricos y descomponerlos más tarde en otros diagramas, la semántica es compleja. Con respecto a la representación del tiempo, no se tiene en cuenta, y no existe un elemento de modelado que lo represente específicamente. Sin embargo dado el carácter abierto que presenta, este se podría simbolizar como un elemento semántico definido por el usuario (como restricción por ejemplo.)

En cuanto a las redes de petri, a medida que se complica el proceso a modelar, el numero de lugares y transiciones crecen demasiado, haciéndolas inaccesibles y en muchos casos imposibles de ser utilizadas para modelar una actividad particular. Esto explica el porqué de las expansiones que sufrieron las redes clásicas en diferentes formas. En relación con el modelado del tiempo, las redes de petri clásicas no lo admiten. Las redes de petri temporales, incorporan el manejo del tiempo aunque si bien esta extensión está acotado a un dominio, siendo difícil de representar otros aspectos del proceso a modelar. Otra característica negativa de éstas es la falta de representación de procesos concurrentes, lo cual no queda definido en ninguna de sus extensiones. Una

combinación de todas las extensiones (jerárquica, tiempo y color) es necesaria para poder representar procesos complejos. Tanto en los statechart como en las redes de petri, el concepto de módulo no existe, dificultando el empleo de modelos genéricos que puedan ser reusados para construir otros modelos. El formalismo DEVS al estar basado en teoría de sistemas, permite representar modelos como sistemas, donde el tiempo se representa explícitamente (función de transición interna). Por otro lado se plantea un enfoque jerárquico y modular en la construcción del modelo, lo cual es una característica deseable y necesaria cuando se trata de modelos muy grandes. Una propiedad importante que tiene DEVS que no presenta los otros paradigmas analizados, es que el modelo se encuentra separado del motor de simulación, característica que suma más ventajas al reuso de componentes. La tabla 2.1 resume la comparación entre los tres paradigmas discutidos en cuanto a las características más sobresalientes de los mismos.

Característica-paradigma	DEVS	Redes de petri	Statechart
Modelado del tiempo	X	PN temporales	semántica
Modularidad	X	-	-
Construcción Jerárquica	X	X	X
Reuso de componentes	X	-	-
Ejecución Concurrente	X	-	X

Tabla 2.1: Comparación entre los formalismos

2.2. Simulación Paralela y Distribuida

El concepto de simulación paralela y distribuida (PADS) no es nuevo sino que se remonta a la década del ’70 donde la necesidad de ejecutar grandes sistemas de simulación obligó a adoptar una visión diferente de manera de superar la falta de recursos (memoria, procesamiento y almacenamiento principalmente) que existían en ese mo-

mento.

Fue así que surge el concepto de simulación paralela (Lin y Lazowska, 1991), en la cual, el sistema a ser simulado es particionado en subsistemas que interactúan a través de un esquema de eventos. El conjunto de subsistemas es simulado por un conjunto de procesos que se comunican a través del envío y recepción de mensajes (o eventos) los cuales tienen asociado una marca temporal indicando el tiempo de simulación en que el evento fue generado. Cada uno de estos subsistemas es ejecutado en un procesador distinto logrando de esta forma ejecutar sistemas de simulación más grandes que los que se podrían lograr en una simulación local (es decir con un único procesador). Pero en esta simulación paralela, el concepto de tiempo global de simulación y de una lista global de eventos presentes en la simulación local, dejan de existir. Cada subsistema tiene su lista de eventos y tiempo de simulación local. Luego cada subsistema es responsable de procesar los eventos en el orden en que los mismos van sucediendo de acuerdo a la marca temporal asignada. En este caso, el subsistema debe procesar primero los eventos con marca temporal más temprana. Esto requiere que la lista de eventos en cada subsistema este ordenada por la marca temporal de los mismos. Inicialmente esta simulación paralela se ejecutaba en máquinas con múltiples procesadores pero más tarde, se comenzaron a utilizar distintas máquinas distribuidas geográficamente. Es entonces cuando la simulación paralela comenzó a llamarse simulación distribuida. Sin embargo la noción de ejecución paralela de subsistema en las distintas máquinas continúa existiendo aún cuando estas estén o no distribuidas geográficamente. Para Fujimoto (2003) la simulación distribuida refiere a distribuir la ejecución de una simple corrida de un programa de simulación a través de múltiples procesadores. El mismo no hace diferencia entre simulación paralela y distribuida dado que considera que dichas diferencias se hacen cada vez menos claras dado al surgimiento de clusters de estaciones de trabajos y el concepto de grid computing.

Una definición más reciente sobre simulación distribuida es la dada por Verbraeck (2004) donde define simulación distribuida como la exposición del comportamiento de un modelo a otros software de simulación a través de un protocolo de simulación de red.

Lo cierto es que generalmente no se hace una diferencia entre simulación paralela y distribuida. Se pueden encontrar estos términos usados indistintamente.

En esta tesis, se considera que la simulación paralela hace referencia a la ejecución simultánea de simuladores, esto es por ejemplo la posibilidad de ejecutar eventos concurrentes pero no se distingue si la ejecución se da o no en máquinas distintas. A su vez se dirá que la simulación es distribuida si la simulación paralela se lleva a cabo en máquinas distribuidas geográficamente.

Aclarado estos conceptos, se continúa analizando cuáles son los objetivos que se quieren lograr con el uso de simulación distribuida. Los mismos pueden resumirse en:

- incrementar la velocidad en la ejecución
- incrementar el tamaño de los modelos, ya que usando simulación distribuida se podría estar ejecutando un modelo que en una sola máquina sería imposible de hacerlo por restricciones de capacidad y tiempo principalmente.
- explotar capacidades especiales de ciertos nodos como ser: capacidad de manejar grandes cantidades de datos, capacidades gráficas, etc.
- integrar diferentes simuladores en un solo entorno de simulación.

Este último objetivo está relacionado al nuevo paradigma de simulación basado en componentes (Verbraeck, 2004), donde se acentúa la construcción modular de simuladores. De esta forma, un simulador es construido a partir de módulos que se relacionan

e interconectan. Algunos ejemplos del uso de este paradigmas son: entrenamientos militares donde los simuladores de tanques de guerra, simuladores de vuelos, generadores de fuertes, y otra variedad de modelos crean un ambiente virtual distribuido donde las personas pueden entrenar planteando situaciones hipotéticas. También este paradigma comenzó a usarse fuertemente en el área de simulación de cadenas de suministro, en simulación de fábricas de manufactura, en simulación en ámbitos tanto sociales como naturales, para predecir o analizar catástrofes.

Si bien, simulación distribuida tiene sus ventajas con relación a simulación local, existen problemas que deben ser tenidos en cuenta a la hora de utilizarla. Estos problemas están relacionados a la sincronización de los simuladores y a la comunicación entre ellos. El primero de ellos, sincronización, es también conocido como administración del tiempo y refiere a la necesidad que la simulación distribuida produzca exactamente los mismos resultados que si es ejecutada localmente. El segundo problema refiere a la interpretación que un módulo hace de los eventos enviados por otros módulos de manera que éstos sean entendidos correctamente provocando los efectos deseados por el diseñador del modelo de simulación.

Con relación al manejo del tiempo, se propusieron diferentes esquemas de administración del tiempo los cuales se conocen como esquema conservativo y esquema optimista. Por otro lado, el problema de la interoperabilidad, es abordado por el standard HLA de la IEEE (IEEE, 2000b) el cual es ampliamente usado por la mayoría de las soluciones propuestas que utilizan simulación distribuida. Este estándar tiene su origen en el standard DMSO HLA 1.3 (DMSO, 1998), luego fue evaluado y estandarizado por el organismo de estandarización IEEE dando origen a una nueva versión conocida como el estándar HLA 1516 (IEEE, 2001). Este estándar provee de interoperabilidad sintáctica a los simuladores pero carece de interoperabilidad semántica. Una nueva versión está actualmente en desarrollo conocida como *HLA Evolved* (Scrudder y otros,

2005) donde se están revisando las modificaciones propuestas al estándar en vigencia. Una de las principales modificaciones planteadas es la posibilidad de cargar los modelos de objetos usados por los simuladores de una manera dinámica y modular (Möller y otros, 2007)(Möller y Lofstrand, 2007)

En las próximas dos sesiones, se analizarán las propuestas existentes para el manejo del tiempo, y se presentará una introducción a la última versión del estándar HLA IEEE 1516.

2.2.1. Esquemas para el Manejo del Tiempo

En simulación, los tiempos son uno de los aspectos que requieren un tratamiento particular.

Considérese la frase: *vamos a simular el proceso de producción desde que el mismo comienza a las 8hs hasta las 16hs, cuando finaliza*. Este intervalo de tiempo de 8hs a 16hs que aparece en la frase, está haciendo referencia al **tiempo físico**, es decir el tiempo en el mundo real del sistema que se quiere simular. No debe interpretarse entonces, que la simulación, debe necesariamente comenzar a las 8hs y finalizar a las 16hs. La ejecución de la simulación puede darse en cualquier momento y no necesariamente durante ese período.

Por otro lado los expertos encargados de construir el modelo de simulación y ejecutarlo dirán frases como esta: *usemos un valor de punto flotante de doble precisión cuyo valor esté en el rango [0.0 8.0] para representar el tiempo en que los eventos se producen*. En esta frase, el tiempo al que se está haciendo referencia es conocido como el **tiempo de simulación**, que indica la manera en que el modelo de simulación representa el tiempo físico. En este caso por ejemplo, se puede estar considerando que una unidad de tiempo de simulación equivale a una hora del tiempo físico. De esta forma,

no importa en que momento la simulación comienza, el simulador interpretará que se comienza en el tiempo de simulación cero representando el inicio del proceso en el mundo real(equivalente al tiempo físico: 8hs) y finalizará en el tiempo 8.0 representando la finalización de dicho proceso (equivalente al tiempo físico 16hs). En la siguiente frase: *la simulación se ejecutó durante 1 1/2 hora, comenzó a las 10 y finalizó a las 11:30hs.* se hace referencia al **tiempo que marca el reloj de pared** ("Wallclock time") que representa el tiempo real en que la ejecución de la simulación es llevada a cabo. Entonces es posible decir: *a las 10hs se comenzó a ejecutar la simulación, cuyo tiempo de simulación comenzó en cero, representando las 8hs del tiempo físico en el que el proceso simulado es realmente ejecutado.*

En algunos casos, el avance del tiempo de simulación está sincronizado con el avance del tiempo del reloj de pared, este tipo de simulación se la conoce como **simulación en tiempo real**.

Cuando se discute sobre la administración del tiempo, se hace referencia principalmente al **tiempo de simulación**. Así en lo sucesivo, cuando se habla de tiempo, se está haciendo referencia al tiempo de simulación.

Ahora bien, es posible formularse la siguiente pregunta: ¿por que es necesario administrar el tiempo? El problema principal que se tiene, y por lo cual se necesita un administrador del tiempo, es que el mundo simulado puede no reproducir correctamente los aspectos temporales del mundo real. Por ejemplo, en el mundo de simulación las demoras pueden producirse por trafico en la red, tiempo de procesamiento de los procesadores intervinientes, demoras en el acceso a los datos almacenado, etc. Es decir cosas que no tienen relación con demoras en el mundo real (rotura de recursos, falta de mercadería, congestión de las rutas, etc). Esto puede producir anomalías durante el proceso de simulación tales como la aparición de la causa después del efecto.

El comportamiento del simulador depende del ordenamiento de los eventos. Así se

puede decir que una simulación es **correcta**, o preserva el **principio de causalidad**, si ejecuta eventos en un **orden temporal estricto**. Para garantizar este orden, cada evento tiene su **marca temporal** ("time stamp") que indica el tiempo de simulación cuando ese evento fue generado. Luego, el orden temporal estricto del que se habla, refiere a la necesidad del simulador de procesar los eventos con marcas temporales crecientes. Fujimoto (1990) define simulación correcta (sin violación de causalidad) como aquella en la que se cumple la restricción de causalidad. La restricción de causalidad está dada por la siguiente definición:

"*la simulación de un modelo consistente de componentes modulares los cuales se comunican exclusivamente a través de sus acoplamientos de E/S, obedece la restricción de causalidad* ***si y solo si*** *cada simulador del componente modular procesa eventos en un orden temporal no decreciente.*"

Cuando se plantea el algoritmo de simulación de eventos discretos, la forma de calcular el siguiente estado se expresa en función del estado actual, el tiempo transcurrido y los eventos recibidos por el modelo. Sin embargo, esta especificación presupone que el comportamiento es generado en un **orden temporal estricto**. No obstante esto no es necesariamente así cuando se considera simulación distribuida. Sea por ejemplo un modelo con dos componentes que actúen estrictamente separados (sin acoplamientos), en este caso, el modelo no se preocupa si el comportamiento de sus dos componentes se da en forma ordenada en el tiempo, ya que esto no afecta a los resultados. Pero esto no es lo que ocurre muy a menudo. Por lo general dos modelos pueden estar actuando independientemente una porción de tiempo, pero después existen dependencias entre ellos. Las dependencias en modelos de eventos discretos son los eventos. Cuando un evento en un componente, directa o indirectamente, influencia eventos sobre otro componente se dice que existe una **dependencia causal** de eventos entre dichos componentes. Un evento E_j depende de un evento E_i si E_i es la causa del evento E_j, si E_i altera el estado

usado por E_j, o si E_i es la causa de un evento E_k y E_j depende de E_k. Para que una ejecución sea adecuada y garantice ser correcta, la generación de eventos debe preservar la propiedad de causalidad, esto es, que la causa siempre preceda al efecto.Esta propiedad, establece que para un evento arbitrario E_j, todos los eventos E_i de los cuales éste depende deben ser procesados antes que el evento E_j. La relación de dependencia es transitiva. Esta dependencia de eventos forma un orden parcial irreflexivo de eventos. Más aún podemos decir que la relación de dependencia no es cíclica y que el tiempo asociado a un evento E_j no sea menor que el tiempo asociado a un evento E_i del cual E_j dependa.

Los algoritmos de simulación secuencial, garantizan causalidad entre eventos ya que generan un orden parcial de eventos de acuerdo al tiempo del evento. En este tipo de simulación, existe una única lista de eventos ordenada por marca temporal y un tiempo de simulación global. Sin embargo la simulación paralela debe explotar el ordenamiento parcial, para encontrar eventos que puedan ser procesados en paralelo. En este tipo de simulación, deja de existir la lista global única de eventos como también el tiempo de simulación global. Ahora cada componente tiene su lista de eventos y su tiempo de simulación local.

Existen dos esquemas de manejo del tiempo en simulación de eventos discretos paralelos: el esquema **conservativo**, que elimina completamente las violaciones a la restricción de causalidad, y el esquema **optimista**, que permite temporariamente la violación de esta restricción, pero luego se detectará la anomalía y se la repara. A continuación se explica con más detalles los esquemas conservativo y optimista para el manejo del tiempo en simulación distribuida.

Esquema conservativo El esquema conservativo fue introducido por Chandy y Misra en 1978 (Chandy y Misra, 1978). Considerando la restricción de causalidad se observa

que el principal problema para un simulador es determinar cuando un evento puede ser ejecutado sin problemas. Esto es, el simulador debe garantizar que no hay ni habrá otro evento con un tiempo anterior al evento que se ejecuta. Un simulador cuando recibe un evento, puede agregarlo en una lista ordenada por tiempo de ocurrencia del evento o marca temporal.

En este esquema, los eventos en los modelos componentes deben ser procesados en orden temporal estricto, y los eventos provenientes de componentes influyentes serán recibidos en orden temporal estricto también. Cuando un modelo componente es influenciado solo por un modelo, se garantiza recibir los eventos en orden temporal estricto y éstos pueden ser procesados tan pronto como se reciben. El problema aparece cuando un componente tiene más de un componente influyente. Considérese el caso mostrado en la figura 2.3 donde el componente J es influenciado por dos componentes: P y Q. En

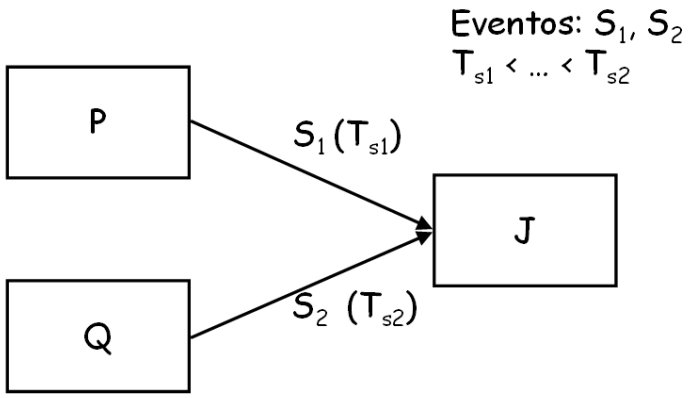


Figura 2.3: Causalidad

esta situación J recibe un evento S_1, de P con marca temporal T_{s1}, y recibe un evento S_2 de Q con marca temporal T_{s2}. Si $T_{s1} < T_{s2}$, J no puede procesar S_2 hasta no recibir el próximo evento de P, ya que no se puede garantizar que P no envíe un evento que esté entre T_{s1} y T_{s2}. Este problema en el esquema de simulación paralelo conservativo,

es crucial cuando la comunicación de eventos no está distribuida uniformemente. Un caso problemático podría ser que J reciba eventos de P pero no reciba eventos de Q por estar este modelo vacío. Entonces J tendrá una lista de eventos sin procesar ya que no puede determinar que no haya algún evento de Q con marca temporal menor a algún evento de P, permaneciendo bloqueado hasta que Q envíe un evento. Pero este hecho nunca sucederá dado que se dijo que Q es un modelo vacío, en esta situación se está en presencia de un abrazo mortal.

El esquema conservativo propone eliminar esta situación mediante la emisión de mensajes nulos, dotando de esta manera al nodo de propiedades de *lookahead*. El *lookahead* (Fujimoto, 1988) es la habilidad de un modelo de predecir que no tendrá salidas por un determinado período en el futuro luego del último evento. Esta propiedad es utilizada por los componentes influyentes para sincronización. Retomando el ejemplo de la figura 2.3 para J,P y Q si se conoce que el lookahead de P es T_l se puede inferir que P no enviará un evento con tiempo menor a $T_{s1} + T_l$. Luego si $T_{s2} < T_{s1} + T_l$ es posible procesar S_1 y luego S_2, ya que P garantiza no enviar un evento con tiempo menor a T_{s2}. Esta garantía está dada por la propiedad de *lookahead*.

El *lookahead* puede ser usado para eliminar los abrazos mortales. Un proceso mira hacia delante (looking ahead) y hace la promesa de no enviar eventos antes de un tiempo dado. Este valor se calcula como sigue:

- un tiempo mínimo estimado para que la próxima salida sea producida, asumiendo que no se reciben eventos (no puede ser menor que el próximo evento interno ordenado dentro del componente)
- Un tiempo mínimo estimado para que la próxima salida sea producida, asumiendo que hay una entrada

En su forma más simple el *lookahead* está dado en forma de demoras de propagación

mínima, el cual representa el tiempo que una salida necesita para propagarse por el sistema y llegar a la entrada.

La performance de un esquema conservativo es fuertemente dependiente de este *lookahead*. Modelos que no tienen una buena propiedad de *lookahead* son difíciles de modelar con este esquema. Ejemplos de tales modelos pueden ser servidores con tiempo de procesamiento cero o que pueden estar vacíos.

Esquema optimista El esquema optimista, a diferencia del conservativo, permite realizar procesamiento de eventos aún con riesgo de violar la causalidad. En este caso, el simulador continúa procesando eventos aún cuando no puede garantizar una ejecución correcta. Como consecuencia un simulador podría estar en un tiempo más avanzado que sus influyentes, y en ese caso podría recibir un evento con un tiempo menor a su tiempo actual de simulación, dicho evento se lo conoce como evento *straggle* o evento desordenado. Esto significa que existe una violación a la causalidad. Por lo tanto debe ser detectada y corregida. La forma de corregirlo es hacer un rollback en el tiempo de simulación de acuerdo al tiempo del nuevo mensaje. El proceso de rollback implica que el simulador debe: (i) restablecer el estado al tiempo anterior al evento *straggle* y (ii) eliminar cualquier salida enviada con un tiempo de simulación posterior al tiempo del evento *straggle*.

Para realizar esta acción el simulador debe guardar información sobre: estado, mensajes de entrada (para poder reingresar las entradas después del rollback) y eventos de salidas para deshacerlos, esta acción es conocida como el proceso de "*matar el evento*". Esta acción afecta a los componentes influenciados: se requiere de mensajes especiales llamados *anti-mensajes* que deben ser enviados para deshacer los efectos de las salidas. El componente que recibe el evento *straggle* es el responsable de "matar los eventos" enviados con marcas temporales posteriores a la marca temporal del evento straggle re-

cibido, luego el componente que recibe un *anti-mensaje* debe actuar de la misma forma que actuó el componente que recibió el evento *straggle*. La necesidad de mantener los estados en este esquema genera una sobrecarga, especialmente en los requerimientos de memoria.

Dado que los componentes necesitan guardar la información de los estados, es importante poder determinar cual es aquella información que no va a ser usada en lo sucesivo, es decir, poder determinar el tiempo mínimo en que no se pueden recibir eventos *straggle*. Luego, toda información correspondiente a un tiempo menor a este tiempo estimado, es información que puede eliminarse con tranquilidad dado que se sabe que el simulador no puede volver a ese estado. Esta información se la conoce como "*Fossil collection*", y al tiempo Tx, el cual es el tiempo mínimo al que se podría regresar por algún mensaje *straggle* se lo conoce como "*horizonte de tiempo*" y es referenciado generalmente como "*global virtual time*" (GVT).

La estrategia descrita corresponde al algoritmo conocido como Time-Warp (Jefferson y Sowizral, 1985) que refiere un simulador de eventos discretos paralelo que implementa un esquema optimista.

2.2.2. Estándar IEEE 1516 High Level Architecture

HLA (High Level Architecture) (DMSO, 1998) es un framework de simulación distribuida propuesto por el departamento de defensa de los Estados Unidos (DMSO: Defense Modelling and Simulation Office) el cual definió una arquitectura para el modelado y simulación de sistemas complejos. Más tarde este framework fue tomado por la IEEE para su estandarización, dando origen al estándar IEEE 1516. HLA es una arquitectura estándar que soporta el reuso de capacidades disponibles en diferentes simuladores, a la vez que posibilita el desarrollo de sistemas de simulación complejos de manera coope-

rativa y distribuida, soportando el desarrollo de simulación basada en componentes, donde los componentes son llamados *Federados*. La arquitectura HLA provee un framework dentro del cual los desarrolladores de simuladores pueden estructurar y describir sus aplicaciones de simulación. En este sentido HLA propone integrar sistemas de simulación diferentes, cada uno con un objetivo determinado, en un sistema mayor, que represente el comportamiento del mismo, sin necesidad de reescribir algún componente o comenzar a crear desde cero el modelo de mayor nivel y su simulador. Un simulador, que cumple las especificaciones HLA, para ser integrado en el framework, se lo llama **federado**. Los federados luego se unen en *federaciones*. HLA utiliza el término **federación** para hacer referencia al simulador del sistema complejo, formado por simuladores de menor nivel llamados **federados** los cuales integran la federación. La definición de HLA abarca tres componentes esenciales (figura 2.4):

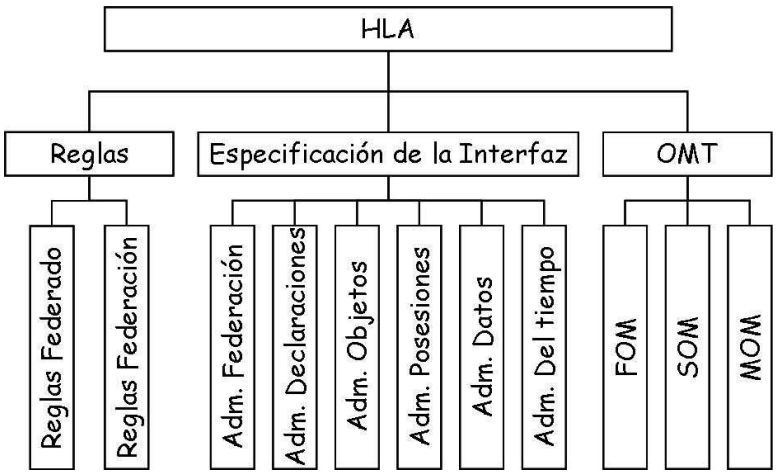


Figura 2.4: Componentes de HLA

A) **Reglas:** son un conjunto de ítems que definen las responsabilidades y relaciones entre los componentes de una federación HLA. Deben ser seguidas por los federados y las federaciones que quieran ajustarse a HLA. Seguir estas reglas asegura una interacción correcta entre los simuladores en una federación.

B) **Especificación de la interfaz:** define la interfase funcional entre federados HLA y la infraestructura de ejecución (runtime infrastructure o RTI) de HLA. Se detallan los servicios que debe prestar el RTI, e identifica las funciones *callback* que debe proveer cada federado. El RTI puede ser visto como el sistema operativo distribuido que provee comunicación y coordinación para los federados.

C) **Patrón del Modelo de Objeto (OMT Object Model Template):** provee un formato común de presentación para el modelo de objetos HLA. Establece el formato de modelos claves como ser: **FOM** Federation Object Model, **SOM**: simulation Object Model y **MOM**: Management Object Model.

Cada federado tiene su modelo de objeto llamado **Simulation Object Model** (SOM) que describe los datos que el federado puede producir o consumir. Cada federación tiene un modelo de objetos propio llamado **Federation Object Model** (FOM) que describe las partes comunes de los SOM, correspondientes a los federados que participan y que serán usados en la federación. El RTI es el software necesario para ejecutar la federación y básicamente consiste en una implementación de la *especificación de la interfase* compuesta por un conjunto de servicios. El RTI provee funcionalidad para comenzar y terminar la ejecución de la simulación, transferir datos entre los simuladores intervinientes, controlar la cantidad de datos que son transferidos y finalmente, coordinar el paso del tiempo de simulación entre los simuladores HLA. Este software está fuera del alcance de la especificación y es suministrado por proveedores de software específicos.

El estándar 1516 está dividido en dos partes:

- 1516.1-2000 IEEE Standard for modelling and simulation high level architecture - Federate Interface Specification

- 1516.2-2000 IEEE standard for modelling and simulation high level Architecture - Object Model Template Specification

El primero de estos documentos *Federate Interface Specification*, describe la interfaz entre el federado y el servicio de software subyacente que soporta comunicación entre federados en un dominio de simulación distribuido. Puntualiza las partes en que se divide la especificación de la interfaz son: *federation management, declaration management, Object management, ownership management, time management, data distribution management*. Cada uno de estos componentes describen detalladamente los servicios que el federado puede pedir al RTI y también los servicios que el federado debe proveer (denominados "callback functions").

El segundo documento, *Object Model Template Specification* describe la sintaxis y el formato para representar la información del modelo de objetos de HLA, incluyendo objetos, atributos, interacciones y parámetros. Forman parte de esta especificación el *FOM* "Federation object model", *SOM* "Simulation object model" y *MOM* "Management object model". El *FOM* tiene como objetivo proveer una especificación común para el intercambio de datos entre federados en un formato estándar. En síntesis el *FOM* establece un contrato del modelo de información que es necesario (pero no suficiente) para alcanzar interoperabilidad entre federados. El *SOM* especifica los tipos de información que un federado individual podría proveer a la federación HLA, y la información que este podría recibir desde otro federado. El *MOM* provee facilidades para acceder a información operativa del *RTI* en tiempo de ejecución. Los federados usan estas facilidades para tener un conocimiento más detallado del funcionamiento de la ejecución de la federación y controlar la ejecución del *RTI* como también de la federación y de los federados individuales. El *MOM* debe ser definido utilizando objetos, interacciones y constructores HLA predefinidos. Todas las clases de objetos, clases

de interacciones, atributos y parámetros deben ser definidos en el archivo *FDD* (FOM document data.)

La figura 2.5 muestra una representación gráfica de una federación (llamada *Una Federación*), y la estructura interna de uno de los federados (Federado A). En esta federación intervienen tres federados llamados: *Federado A, Federado B* y *Federado C*. Estos federados comparten el *FOM* y se comunican unos con otros a través del *RTI*. El componente FedExec1, corresponde a la ejecución de la federación, y es creado cuando se invoca el servicio *createFederationExecution* sobre el RTI. Este servicio puede ser invocado por cualquiera de los federados o aún por algún componente externo a la federación. Durante la ejecución del RTI varias federaciones pueden estar activas pero no es posible compartir federados. Cada uno de los federados debe identificar correctamente en qué federación va a participar. En la figura 2.5, se muestra la estructura interna del

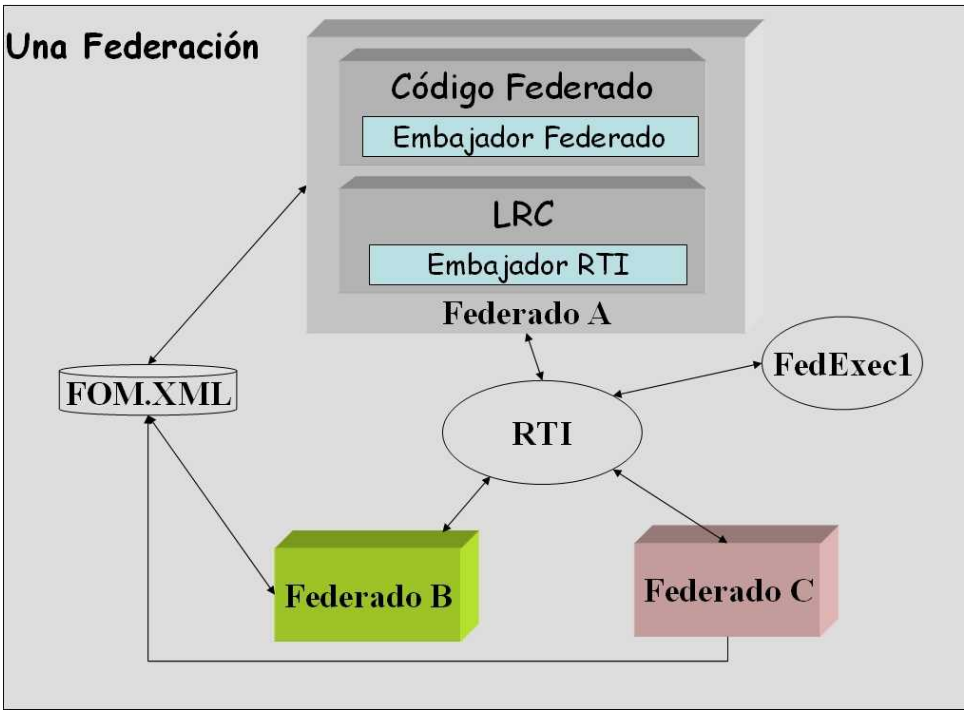


Figura 2.5: Esquema de una federación HLA

federado "*Federado A*"

De acuerdo a la definición dada por HLA, *un* Federado *es una aplicación que puede ser o está actualmente acoplada con otras aplicaciones de software bajo un FDD (Federation object model Document Data) y un RTI (Run Time Infrastructure).* Así un federado está compuesto por dos elementos esenciales: el **LRC** (local run time Component infrastructure) y el **Código Federado** (código que implementa la funcionalidad del modelo).

El **LRC** tiene como componente el *EmbajadorRTI*, el cual es el encargado de la comunicación entre el RTI y el federado. Este componente debe invocar los servicios sobre el RTI correspondientes a aquellas funciones definidas en "*HLA interface specification*", como ser por ejemplo subscribir al federado a una interacción, publicar interacciones y solicitar permiso para avanzar el tiempo, entre las más comunes.

El segundo componente en un Federado es el *Código federado*, el cual representa la simulación llevada a cabo. Este componente tiene un *Embajador Federado* el cual implementa las funciones "*callback*" que serán invocadas por el *RTI*. Por ejemplo cuando el RTI envía una interacción al federado, estará invocando funciones sobre el *Embajador Federado* (en este caso se trata de la función *recibeInteraction* definida en la especificación), el cual debe tratar de una manera particular dicho requerimiento. Ejemplos típicos de la invocación de funciones son las que realiza el RTI para dar permiso al federado para avanzar el tiempo de simulación (timeAdvanceGrant), enviar una interacción (recibeInteraction), informar sobre la actualización en los atributos de un objeto (attributeValueUpdate), etc.

De esta forma, todas las interacciones desde el RTI hacia el federado son manejadas por el *Embajador Federado*, mientras que todas las interacciones desde el federado hacia el RTI son controladas por el *Embajador RTI*.

Como se dijo anteriormente, el RTI es software que está fuera de la especificación HLA pero que debe estar de acuerdo a la misma. RTI provee los servicios de software necesarios para soportar simulación con HLA ya que:

- Brinda interoperabilidad y portabilidad
- Permite separar simulación de comunicación
- Facilita la construcción y destrucción de federados
- Soporta la declaración y administración entre federados
- Asiste en la administración del tiempo del federado
- Provee comunicación eficiente a grupos lógicos de federados

El RTI es un componente que está especificado en HLA, pero no forma parte del framework, es decir, la especificación establece cuáles deben ser los servicios provisto por RTI, pero no especifica cómo deben ser implementados. Este componente permite que las federaciones interactúen. Pueden encontrarse distintas implementaciones de RTI que cumplen con las especificaciones HLA. En su gran mayoría son versiones comerciales. Particularmente para el desarrollo de esta tesis se seleccionó una herramienta de software libre llamada poRTIco (Pokorny y otros, 2008).

Los principales aspectos del framework que define la especificación HLA, se pueden organizar en los siguientes temas a ser considerados:

- Comunicación entre los federados
- Administración del tiempo en HLA
- Orden de los mensajes y marca temporal
- Avance del tiempo lógico

- Cálculo de LBTS
- Lookahead
- Lookahead cero y eventos simultáneos
- Procesamiento de eventos optimista.

Comunicación entre los federados. Los federados en una federación no se comunican directamente unos con otros sino que lo hacen a través del RTI. Existen dos formas en que los federados pueden comunicarse: (i) a través del envío de interacciones y (ii) a través de la actualización de valores de atributos de objetos. Tanto las interacciones intercambiadas como los atributos actualizados, deben ser parte del FOM. Como se indicó anteriormente, el FOM representa un contrato entre los federados que establece qué datos van a ser compartidos durante la ejecución de una federación. Cuando los federados se unen a la federación, se asume que éstos conocen y aceptan dicho contrato.

El primer tipo de comunicación, el envío de interacciones, es una comunicación que se da entre dos federados a través del RTI. Esta comunicación es no persistente, esto quiere decir que una vez recibida la interacción la misma es tratada por el federado receptor y descartada. Las interacciones pueden tener parámetros.

El segundo tipo de comunicación se da cuando se comparten atributos de objetos. Esta comunicación se considera como una comunicación entre objetos de distintos federados. La comunicación es persistente dado que el valor del atributo permanece aún después que el federado haya procesado la información solicitada.

Los mecanismos para publicar y subscribir interacciones y atributos están descriptos en detalle en el punto "*Declaration management*" de la especificación de la interfase.

En cualquiera de los dos casos, se utiliza el mecanismo "suscribir/publicar". Un federado puede subscribirse a las interacciones y a los atributos que le sean de interés,

y publicar las interacciones y los atributos que serán generados como consecuencia de su actividad. Luego, cuando un federado publica una interacción, el RTI se encarga de enviarla a todos aquellos federados que están subscritos a la misma. Lo mismo sucede con los atributos: un federado dueño de un objeto, modifica un atributo de dicho objeto, como consecuencia, el RTI informa a todos aquellos interesados de la modificación que se produjo. Se debe notar que sólo "*el dueño del objeto*", puede en cualquier momento modificar el atributo de un objeto. Existen mecanismos que le permiten a un federado solicitar al RTI la tenencia de un objeto, así como también liberarlo para que otro federado pueda poseerlo. Estos mecanismos están definidos en la sección "*Ownership management*" de la especificación de la interfase.

Administración del tiempo en HLA El tiempo de simulación en HLA es representado como puntos sobre el eje del tiempo global de la federación. En un instante de la ejecución, un federado estará sobre un punto de este eje, referido como el tiempo del federado.

En HLA, como en otros simuladores, las anomalías se eliminan asignando una marca temporal o "*time stamp*" a cada evento, y asegurando que los eventos son entregados al federado en forma ordenada de acuerdo a su marca temporal. Además el servicio de administración del tiempo asegura a un federado no recibir eventos correspondientes a su pasado. Es decir el federado tiene su tiempo local, los eventos que llegan al federado deben tener marcas temporales mayores a ese tiempo local, y nunca menores a este. Una meta clave de HLA es la interoperabilidad y el reuso de simuladores. Para soportar interoperabilidad, el servicio de administración del tiempo debe permitir simular usando diferentes mecanismos de administración del tiempo interno, para ser combinados en la ejecución de una federación simple. Un aspecto importante para la interoperabilidad es la transparencia en la administración del tiempo, lo cual requiere que el mecanismo

interno para administrar el tiempo de un federado no debe ser visto por los otros federados.

Cada federado puede tener su propio y diferente mecanismos de administración del tiempo. Es posible dar la siguiente clasificación de federados de acuerdo a cómo administra el tiempo:

- **Dirigido por eventos (event driven)**: el federado procesa los eventos locales y los eventos generados por otros, ordenados de acuerdo a sus marcas temporales. El tiempo del federado típicamente avanza al tiempo de la marca temporal del evento procesado.
- **Pasos del tiempo (time stepped)**: el federado avanza el tiempo de simulación en una cantidad fija llamada step (paso). El simulador no avanza a la próxima etapa si no se realizaron todas las actividades de la etapa actual.
- **Simulación de eventos discretos paralelos (parallel discrete event simulation)**: los federados que ejecutan sobre sistemas de múltiples procesadores deben ser sincronizados internamente usando protocolos optimista o conservativos.
- **Dirigido por el tiempo del reloj real (Wallclock time driven)**: este mecanismo se utiliza para aquellos simuladores donde su tiempo de simulación es derivado del tiempo del reloj real. El tiempo que demanda la simulación debe coincidir con el tiempo que transcurre cuando ese proceso se ejecuta en la realidad. Los federados que usan este mecanismo, no requieren necesariamente que los eventos sean procesados en un orden de marca temporal estricto.

La administración del tiempo en una federación debe ser realizada por el federado y el RTI. Una pregunta que surge de esto es: ¿qué funcionalidades deben estar en el RTI

y cuáles en cada federado?

Dado que un federado tiene que procesar eventos internos y eventos producidos por otros federados, necesita poder ordenarlos de acuerdo a la marca temporal que cada uno de ellos tiene. También se deben tomar precauciones para que un federado no reciba eventos en su pasado. Estas tareas son llevadas a cabo por el servicio de administración del tiempo de HLA a través de:

- Un servicio de entrega de mensajes en un orden de marca temporal (TSO).
- Un protocolo donde el federado explícitamente solicite el avance de su tiempo interno, y el RTI provee un permiso para realizar el avance cuando pueda garantizar que no arribarán eventos con tiempo menores.

La figura 2.6 muestra los servicios para el manejo del tiempo provisto por el RTI a los federados. El RTI mantiene dos listas, una cola (FIFO) donde los eventos están ordenados de acuerdo al orden de llegada al RTI y otra cola (TSO time stamp order) donde los eventos se ordenan por sus marcas temporales. Estos eventos serán entregados a los federados de acuerdo al tipo de servicio que estos requieran. Existen federados que recibirán los eventos de la cola FIFO y otros que recibirán los eventos de la cola TSO.

Orden de los mensajes y Marca temporal. Los mensajes que son recibidos por el RTI pueden ordenarse de dos maneras: (i) por el orden de llegada (RO), en cuyo caso están listos para ser enviados a los destinos, o (ii) por orden de marca temporal (TSO), en cuyo caso, no van a ser entregados a sus destinos hasta que el servidor no pueda determinar que no exista un evento con una marca temporal menor que puede ser recibido en un futuro inmediato para ese destino. Para lograr esta propiedad, el RTI debe calcular un límite inferior de las marcas temporales de los mensajes futuros.

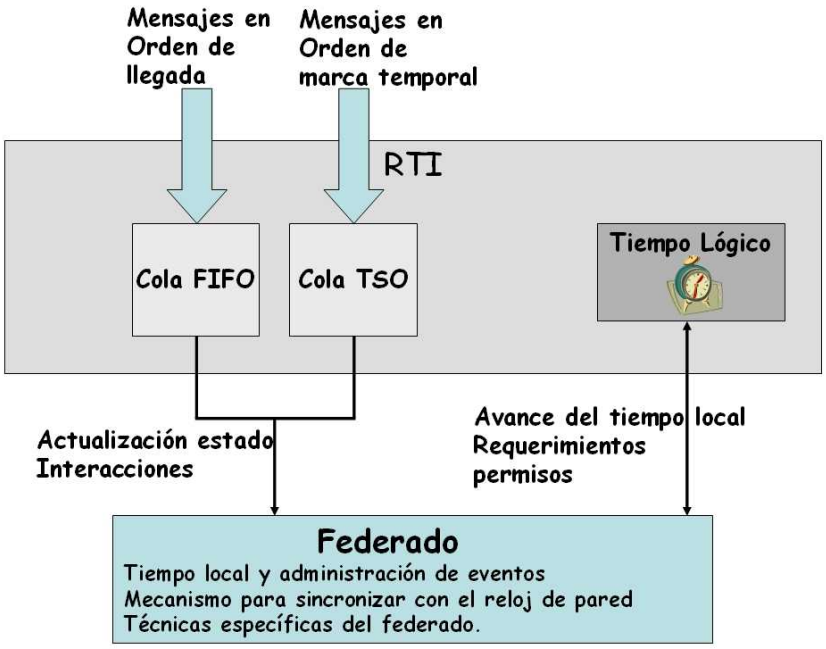


Figura 2.6: Manejo del tiempo en HLA

La marca temporal es asignada al mensaje por el simulador que lo genera y es entregado al receptor en orden no-decreciente de marca temporal. Dado que el RTI no puede entregar mensajes TSO inmediatamente, esta estrategia genera mayor latencia en los mensaje que la RO.

Los federados pueden recibir cualquier tipo de mensajes tanto TSO o RO. En el momento de asociarse a una federación, el federado informa al RTI qué tipo de servicio va a requerir.

Los mensajes que tienen la misma marca temporal, correspondiente a eventos simultáneos, son entregados al federado en un orden arbitrario; el federado que recibe estos mensajes, puede almacenarlos temporalmente en un buffer y ordenarlo. Puede suceder que el federado, incluya un campo en la marca temporal del mensaje cuando éste es generado para ordenar los eventos simultáneos. En este caso, debe existir en la especificación de la federación, el formato y el significado del campo de marca temporal

así como los valores y duración del tiempo lógico. Entonces, la federación puede por ejemplo especificar la representación del tiempo usando un tipo de datos abstracto. La especificación del formato de la marca temporal debe incluirse en el OMT. A veces la federación debe indicar también ciertas operaciones tales como comparación de los valores de marca temporal.

Avance del tiempo lógico. El RTI debe garantiza a los federados no recibir eventos con marca temporal menor a su tiempo lógico actual. Para cumplir con esta garantía, se establece un contrato entre el federado y el RTI de acuerdo a cómo avanzar el tiempo lógico. Así, cada vez que un federados necesita avanzar su tiempo lógico dado que proceso todos los eventos en el tiempo actual de simulación, es necesario que solicite permiso al RTI para hacerlo. El federado no podrá avanzar su tiempo lógico hasta que no reciba del RTI el permiso para hacerlo. Este protocolo es central en el manejo del tiempo en HLA.

El ciclo de administración del tiempo consiste de tres etapas: (i) el federado envía al administrador del tiempo su requerimiento de avance del tiempo (por ejemplo a traves de la invocación del servicio "*TimeAdvanceRequest*"), (ii) El RTI entrega mensajes (tal vez cero) al federado, invocando servicios definido en el federado como puede ser "*reflectAttributeValue*" para entregar nuevos valores de atributos de objetos o "*receiveInteraction*" para entregar eventos de interacción. (iii) el ciclo se completa cuando el RTI invoca el proceso definido en el federado "*timeAdvanceGrant*" que autoriza al federado avanzar su tiempo lógico.

Existen dos mecanismos definidos en la especificación de la interface de HLA, que pueden ser usados por el federado para solicitar permiso para avanzar su tiempo lógico: (i) *TAR "time advance request"* y (ii) *NER "next event request"*. *TAR* es mejor para federados que usan internamente un mecanismo de tiempos en etapas, mientras que

NER es la primitiva preferida para federados dirigidos por eventos. Sin embargo no existen restricciones en cuál primitiva deben usar los federados. Estas primitivas se definen como sigue:

TimeAdvanceRequest(t) : el federado invoca este servicio para solicitar que su tiempo lógico avance al valor *t*. Todos los mensajes RO que se encuentren en la cola del RTI y todos los mensajes TSO con una marca temporal menor a t son entregados al federado después de la invocación de este servicio. Cuando no existan más mensajes a entregar, y se garantice que ningún federado genere en el futuro un evento para el federado que invocó el servicio, entonces el RTI llama al proceso *timeAdvanceGrant* del federado dándole la autorización para que avance su tiempo al valor *t*.

NextEventRequest(t) : un federado dirigido por eventos, típicamente invoca este servicio cuando haya terminado todas las actividades de simulación en el tiempo lógico actual, y está listo para avanzar al próximo. El parámetro t indicado en la invocación es el tiempo al cual el federado quisiera avanzar. Típicamente t es el tiempo del próximo evento dentro del conjunto de eventos pendiente del federado. Después de la invocación del servicio, el RTI envía todos los eventos RO que están en cola. Si no hay mensajes TSO con una marca temporal menor o igual a t y ningún evento será recibido en el futuro, el RTI invoca el proceso del federado "**timeAdvanceGrant**" indicando que su tiempo debe avanzar a *t*. En otro caso el RTI entregará el próximo mensaje TSO más pequeño destinado para el federado (con un time stamp t‘ donde t‘ $<$ t) y todos los otros mensajes con time stamp t‘. A continuación el RTI invoca el servicio **timeAdvanceGrant(t‘)**, con parámetro t‘.

El ordenamiento de las marcas temporales y el avance del tiempo, forman el com-

ponente central de la arquitectura de administración del tiempo en HLA.

Se diseña la interfaz de manera tal que el RTI cuente con información necesaria para implementar eficientemente las primitivas de administración del tiempo, sin embargo no existen restricciones sobre un protocolo determinado de sincronización que deba ser usado.

La interfaz representa un tipo de contrato entre el RTI y el federado en relación a la marca temporal de los mensajes que los federados generarán durante la ejecución. Para federados que operan con procesamiento paralelo con sincronización optimista, existen primitivas adicionales.

Cálculo de LBTS. Un aspecto clave en la implementación del servicio de administración del tiempo en RTI es el cálculo del *LBTS ("lower bound time stamp")* para cada federado. Para un federado i su $LBTS_i$ es el límite inferior de las marcas temporales de los mensajes que pueden ser recibidos más tarde en la ejecución, por dicho federado. El RTI tiene dos tareas principales: (i) entregar los eventos en orden de marca temporal y (ii) asegurar que no se entreguen mensajes al federado con una marca temporal menor a su tiempo actual. Una vez que el RTI calcula el $LBTS_i$ puede entregar todos los mensajes con tiempo menor al $LBTS_i$. Luego si el RTI impide que el federado avance su tiempo por arriba de $LBTS_i$, puede garantizar que el federado no recibirá eventos correspondientes a su pasado.

Para calcular el LBTS de cada federado el RTI debe tener en cuenta:

- La marca temporal más pequeña de los mensajes TSO y el tiempo lógico de los federados que podrían ser generadores de eventos en el futuro. El tiempo lógico actual de un federado es un límite, ya que ningún federado puede generar evento con un tiempo menor a su tiempo lógico actual.

- Las marcas temporales de los mensajes en el RTI y de la red de interconexión

Una federación HLA, puede incluir federados con distintos servicios de entrega de mensajes, esto es, algunos federados requieren el servicio de TSO y otros de RO. Para poder manejar estos distintos tipos de federados, el RTI debe determinar cuales de estos federados participan del cálculo del LBTS y aquellos que requieren el resultado de este cálculo.

En HLA, cada federado tiene una señal ("flag") booleana que indica al RTI su estado con respecto al cálculo del LBTS: la señal de "*time constrained*", y la señal "*time regulating*". Un federado con la señal "*time regulating*" es uno que es capaz de generar mensajes TSO y entonces debe ser considerado en el cálculo del LBTS. Un federado con una señal *time constrained* es uno que recibe mensajes TSO y por lo tanto requiere el resultado de calcular LBTS. El servicio de administración del tiempo en HLA provee primitivas que le permiten a los federados activar o desactivar estas señales (ponerlas en "on" u "off"). Si un federado no tiene especificado ninguna de estas señales cuando envía un mensaje TSO, entonces el RTI lo convierte inmediatamente a un mensaje RO.

Lookahead El "lookahead" es una estimación de un federado, indicando que no existirán eventos de salida hasta un tiempo igual a la suma del tiempo actual del federado más el "lookahead" estimado. Por ejemplo, si el tiempo actual del federado F es 10 y su lookahead es 5, este puede estimar que en el tiempo $(10 + 5($, este federado no emitirá ningún evento de salida. Esto le permite a los otros federados avanzar hasta ese tiempo (15) sabiendo que no van a existir eventos de este federado que sean menores a 15. Esta propiedad puede ser difícil de incorporar en algunos tipos de simuladores, pero es muy importante para garantizar los requerimientos de simulación cuando se necesita ordenamiento de los eventos de acuerdo a su marca temporal. Este valor no puede ser determinado por el RTI, dado que es dependiente del simulador. A continuación se dan

algunas ideas de los elementos que pueden ser tenidos en cuenta para poder derivar el lookahead:

- Limitaciones físicas referentes a cuan rápido un federado puede reaccionar a un evento externo.
- Limitaciones físicas referentes a cuan rápido un simulador puede afectar a otro simulador.
- Tolerancia a inexactitudes temporales: por ejemplo si un simulador con tiempo T genera un evento de salida con tiempo $(T + 1)$ y esto no tiene implicancias sobre los resultados de simulación. Entonces el simulador podría organizar sus eventos con 1 segundo más en el futuro dando un "lookahead" de esta cantidad.
- Simulación en etapas del tiempo: en simuladores con avance del tiempo en etapas (step), el "lookahead" por lo general es el tamaño de la etapa o paso.
- Comportamiento no dirigido (que no se tiene poder para cambiarlo o influir sobre éste)

Lookahead cero y eventos simultáneos. Cuando el RTI envía el mensaje "*TimeAvanceGrant*" a un federado, se garantiza que el mismo ha recibido todos los eventos hasta el tiempo *t*. Esta garantía es necesaria para el federado ya que puede organizar los eventos en orden de marca temporal hasta el tiempo *t*. Esta garantía no la puede mantener el RTI si el federado tiene un "lookahead" igual a cero, ya que el federado podría enviar mensajes de salida con tiempo igual a t con lo cual no se mantendría la garantía.

En los primeros comienzos se restringió a los federados a tener un "lookahead" positivo. Dado que esta restricción es fuerte, y con el propósito de permitir federados

con "lookahead" cero, se agregaron dos nuevos servicios: **NERA** (next event request available) y **TARA** (time advance request available). En este caso el RTI cuando envíe la orden de avance del tiempo (time advance grant) al tiempo t no garantiza que el federado haya recibido todos los eventos con tiempos exactamente iguales a t. Pero por otro lado, cuando el federado recibe la autorización de avance del tiempo, no puede (se le prohíbe) enviar eventos con tiempo t, aún cuando su lookahead sea cero.

En simulación distribuida, el RTI tiene en cuenta el tiempo más pequeño de los eventos sin procesar de todos los federados pertenecientes a una federación. Para evitar abrazo mortal esperando el permiso para avanzar el tiempo, el federado debe garantizar con respecto a la marca temporal, al RTI lo siguiente:

- El federado no puede generar eventos de salida con una marca temporal menor o igual al resultado de la suma de su tiempo actual más su lookahead. Aún cuando su lookahead sea cero, el federado no puede generar eventos con marca temporal igual a su tiempo lógica actual.

- Si un federado tiene pendiente un **NER** con un parámetro t, este garantiza que todo los nuevos mensajes generados por este federado tendrán una marca temporal mayor o igual a t más el lookahead, (o estrictamente mayor a t si el lookahead es cero). Si el federado tiene una llamada **NERA** pendiente, cumple con la misma garantía excepto que puede generar eventos de salida con tiempos iguales a t.

- Si el federado tiene una llamada TAR pendiente con parámetro t el federado debe garantizar incondicionalmente que sus nuevos mensajes de salida tendrán una marca temporal mayor o igual al tiempo lógico actual más su lookahead (si su lookahead es cero, la marca temporal debe ser estrictamente mayor al tiempo lógico). Para el caso que la llamada haya sido **TARA**, la misma garantía debe cumplirse pero puede generar eventos con marca temporal iguales a t si el lookahead es cero.

Estas reglas forman un contrato entre el federado y el RTI. Este contrato es importante para el RTI ya que le permite determinar la mínima marca temporal que puede generar un federado, dato que usará para calcular **LBTS**.

Procesamiento de eventos optimista. Se requieren las siguientes capacidades para permitir este tipo de computación:

- El federado debe ser capaz de recibir mensajes TSO antes que el RTI le pueda garantizar que no serán enviados eventos con tiempos menores. En HLA esta capacidad es provista por el servicio "*flush queue request*" que obliga al RTI a enviar todos los eventos en su buffer a la cola interna del destino.

- El federado debe ser capaz de cancelar mensajes previamente enviados. Esto se lleva a cabo a través de la primitiva de HLA "*retract*". Cuando el federado recibe el requerimiento de cancelación, es responsabilidad de este hacer roll back y cancelar sus mensajes enviados si es necesario.

- Los federados optimistas deben ser capaces de calcular un límite inferior del tiempo lógico al que pueden hacer roll back. Este límite inferior es llamado "*Global Virtual Time* " (GVT), este límite permite a los federados optimistas ahorrar memoria y ejecutar operaciones que no van a ser vueltas atrás. En el algoritmo time warp, el rollback es causado cuando se reciben eventos con tiempos menores al tiempo actual o cuando se reciben eventos de cancelación, entonces el $LBTS$ del federado provee información necesaria para el cálculo de GVT.

Una propiedad importante es que el federado que usa el algoritmo "time warp", es capaz de conectarse al RTI sin necesidad de ningún otro federado. El RTI permite que los federados optimistas intercambien mensajes entre sí, a la vez que garantiza

que los mensajes optimistas sean entregados a federados conservativos. Más aún, no deben ser intercambiados mensajes para el cálculo del GVT ya que el RTI envía toda la información necesaria para que cada federado calcule su GVT. Otra ventaja significativa es que deben realizarse pequeñas modificaciones al algoritmo de administración del tiempo conservativo que ya está especificado en el RTI.

Un problema de los simuladores optimistas es el control de la velocidad a la que estos trabajan. Si estos federados trabajan muy rápido y procesan demasiados eventos, la ejecución será ineficiente porque requerirá de muchos rollbacks, lo cual es actualmente una responsabilidad del federado.

Una implementación de RTI particular: poRTIco poRTIco (Pokorny y otros, 2008) es un framework de uso libre y open source que implementa RTI de acuerdo a la especificación IEEE 1516. Este framework cuenta con la certificación de la IEEE. Fue desarrollado por Tim Pokorny, Michael Fraser y Lance Burns. Actualmente está en período de prueba. La elección de este software se hizo teniendo en cuenta principalmente que es una herramienta de software libre. Las versiones comerciales de RTI son costosas y no se tiene acceso al código fuente, necesario para analizar y entender el funcionamiento. La característica de ser software libre, brinda la posibilidad de modificarlo en casos particulares contribuyendo a su mejora beneficiando a otros usuarios.

2.3. Conclusiones

En este capítulo se presentan los diferentes paradigmas de simulación disponibles actualmente que pueden ser aplicados a la modelización de procesos de negocios en empresas, así como la aplicación de simulación distribuida para dar soporte a modelos complejos. A partir del análisis, el formalismo DEVS se selecciona para ser utilizado en

esta tesis por las ventajas que éste presenta. Finalmente se sintetizan los conceptos que deben ser tenidos en cuenta en simulación distribuida y se hace una breve referencia del estándar HLA el cual especifica la infraestructura de ejecución de sistemas de simulación distribuida y heterogénea. Principalmente se hizo hincapié en el manejo del tiempo, uno de los problemas más importantes a resolver por los simuladores participantes de la simulación distribuida.

3

Arquitectura DE^{2}M para la Construcción de Modelos de Empresa Ejecutables y Distribuidos

En este capítulo se presenta la arquitectura propuesta para el desarrollo y ejecución de los modelos de empresa ejecutables y distribuidos. La misma es una arquitectura en capas que permite ocultar el proceso de simulación al experto en procesos de negocios. Se explica cada componente de la arquitectura y el conjunto de pasos a seguir, propuestos en esta tesis, para el desarrollo de los modelos con la misma.

3.1. Arquitectura DE^{2}M

La arquitectura que se presenta, representa la conceptualización de una herramienta de software para dar soporte al desarrollo y ejecución de los modelos DE^{2}M.

Una arquitectura de software (Shaw y Garland, 1996), es la estructura de un sistema, expresada a través de una colección de *componentes* computacionales junto con la descripción de las interacciones entre estos componentes, conocidos como los *conectores*.

El objetivo de esta arquitectura es la definición conceptual de una herramienta de software que brinde soporte en dos principales actividades: (i) representar el conocimiento de la empresa construyendo el modelo de empresa (EM) con un lenguaje amigable

y una interfaz gráfica apropiada, (ii) analizar el comportamiento de los proceso de la empresa a través de su simulación.

En otras palabras, esta arquitectura sustenta el proceso de analizar la empresa tanto en sus aspectos estáticos como también dinámicos. Más específicamente es posible dar respuesta a las siguientes preguntas: ¿que tareas hacen uso de un determinado recurso? ¿que tareas generan un dado recurso?¿que recursos participan en la ejecución de una tarea para que la misma alcance un determinado estado? ¿cuales son las precondiciones para ejecutar la tarea T?¿que tareas preceden a la tarea T?. Además, se abre la posibilidad de generar escenarios What-if? para representar situaciones hipotéticas, las cuales permitirían analizar cuál sería el comportamiento del sistema bajo dichas circunstancias.

Esta propuesta está orientada principalmente a la definición y desarrollo de modelos de empresa (EM) y su posterior análisis empleando simulación, sin que el modelador deba implementar el modelo de simulación (SM). Esta característica que permite construir un EM, con capacidades de simulación parece no tener demasiada significación, sin embargo influye considerablemente a la hora de definir y analizar los procesos.

Para explicar el esquema de modelos propuesto, considérese el ejemplo siguiente: se quiere modelar la actividad realizada por un empleado, que recibe órdenes de cliente y debe armar el pedido para enviar. A partir de este ejemplo el experto en negocios definirá un modelo de esta actividad como el mostrado en la figura 3.1(a) donde, una tarea **Preparar Pedido** es definida relacionada con recursos: **Stock**, **OC** (Orden Cliente), **Empleado Depósito** y **Pedido**. Cada uno de estos recursos está definido con diagramas de estados donde se representa el cambio de estado de participar en las tareas. En la figura 3.1 (b) se muestra el ciclo de vida correspondiente al recurso OC. Así por ejemplo, el recurso *OC* pasa del estado *sinProcesar* al estado *procesada* al participar de la tarea **Preparar Pedido**.

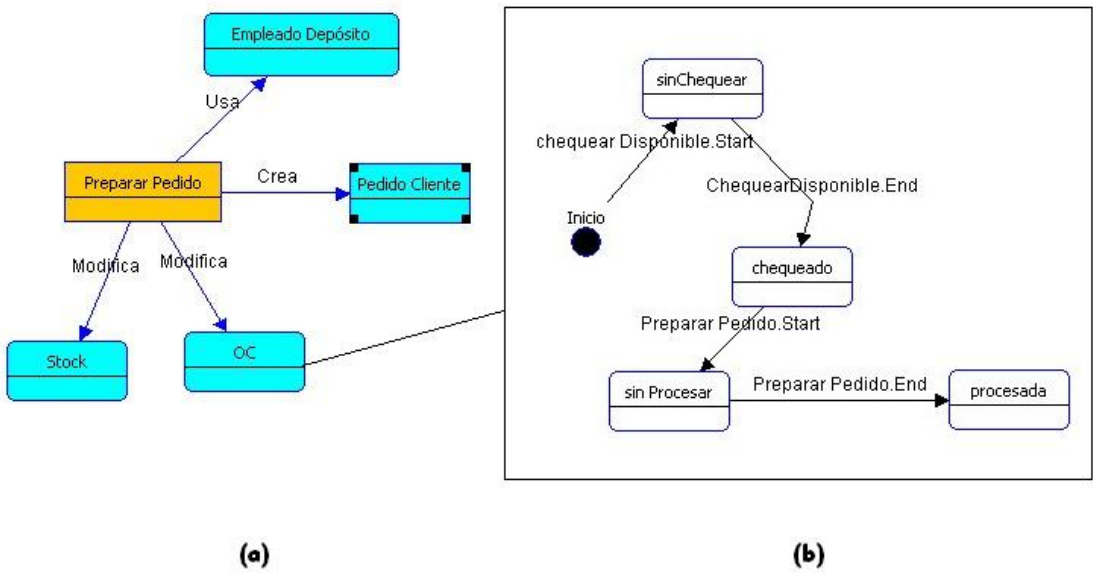


Figura 3.1: (a) Tarea: Preparar pedido. (b) ciclo de vida Recurso OC

Si ahora se construye un modelo de simulación de esta actividad, un experto en simulación creará el modelo que aparece en la figura 3.2 donde, la actividad *Preparar pedido* es modelada como un servidor con una cola de trabajos en espera. Este servidor recibe los eventos que están en el *Buffer* para ir procesándolos en orden de almacenamiento. Las OC se modelan no como recursos, sino como eventos (flecha dirigida en la figura 3.2) que el servidor puede tratar. Se introduce también un generador aleatorio de OC llamado *Generador OC* que es el encargado de alimentar el modelo de simulación. Esto quiere decir que el generador produce los eventos correspondientes a las OC que serán tratados por el servidor. El generador debe manejar un tiempo que indique el período en el cual el evento será producido. Los eventos generados, se dirigen al Buffer donde esperan ser procesados por el servidor. También se define un tiempo aleatorio de servicio, el cual representa el tiempo en que una OC será procesada por el servidor *Preparar Pedido*. Finalmente cuando el servidor termina de procesar el evento OC, envía

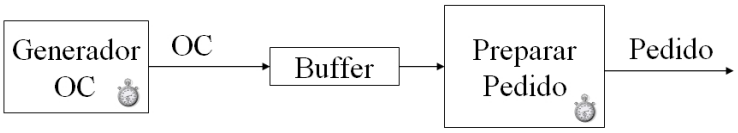


Figura 3.2: Modelo de simulación de la tarea preparar Pedido

el evento de salida *Pedido* hacia otros servidores y obtiene del *Buffer* el próximo evento a tratar.

Obviamente, para un experto en simulación, el modelado de esta actividad es una tarea trivial que no demanda esfuerzos. No obstante, este escenario plantea las siguiente cuestiones: ¿puede el experto de negocios darse cuenta tan rápidamente que esa actividad se representa con un modelo de un servidor con una cola? Éste ¿conoce que es una cola? ¿conoce cómo representar números aleatorios? ¿es capaz de descubrir que la OC debe modelarse como eventos? ¿está dispuesta la organización a incorporar nuevo conocimiento para poder representar sus procesos como modelos de simulación?.

Del análisis de este ejemplo se puede inferir que ambos modelos expresan vistas diferentes y por lo tanto, los conceptos que se utilizan para definir una y otra son diferentes, y requieren diseñadores con diferentes habilidades. Es evidente que el lenguaje que se utilice influye considerablemente sobre que tan fácil le resulta al diseñador construir un modelo que integre ambas vistas.

Algunas propuestas, plantean la construcción de SM usando lenguajes orientados a negocios tales como SIMPROCES (CACI, 2006), o usando lenguajes similares al inglés como es el caso de SIMSCRIPT (Rice y otros, 2005) pero la diferencia principal de estas herramientas de simulación con la propuesta de este trabajo es que las primeras proponen construir un SM y no un modelo conceptual de la empresa. En este caso, el diseñador se ve obligado a representar actividades tales como *contar el número de órdenes recibidas*, cuando en la realidad esta actividad no es ejecutada, pero para el

caso de un SM es fundamental representarla para luego poder obtener estadísticas que permitan el análisis cuantitativo del modelo. Se puede decir en esta situación, que el conocimiento de los procesos de negocio se encuentra combinado con el conocimiento sobre simulación. Esta es una de las razones por las cuales se propone construir un modelo conceptual de la organización, de manera tal que el experto en negocios haga lo que mejor sabe hacer, expresando su conocimiento con un lenguaje adecuado y suficientemente expresivo.

En la elaboración de esta propuesta se optó por ocultar el proceso de simulación y orientar principalmente el resultado al experto de negocios. Es por esta razón que se seleccionó la arquitectura en capas (Gamma y otros, 1996), donde el proceso de simulación es interpretado como un servicio. Este estilo arquitectónico, permite representar el sistema organizado jerárquicamente donde cada capa provee un conjunto de servicios (figura 3.3). Las capas superiores son vistas como clientes de las capas inferiores, las cuales son los servidores. En esta jerarquía, la capa superior, es la capa visible del sistema mientras que las otras están ocultas excepto a las capas que están inmediatamente sobre y debajo de ésta. El estilo arquitectónico permite organizar el sistema basado en servicios donde cada capa se presenta como un conjunto de servicios que brinda a la capa inmediatamente superior.

Particularmente se emplearon dos capas: una para el desarrollo del modelo conceptual y otra para el desarrollo y ejecución del modelo de simulación. Éstas corresponden a: la capa de modelado conceptual (*Conceptual model layer*) y la capa de simulación (*Simulation layer*). La primera de éstas es la capa con la cual el usuario interactúa. Las funciones ofrecidas por la misma están relacionadas con el desarrollo del modelo de empresa, usando un lenguaje orientado a negocios. El desarrollo del modelo de empresa se realiza con un vocabulario compuesto por conceptos tales como: tareas, recursos, relaciones entre estos, condiciones, etc.

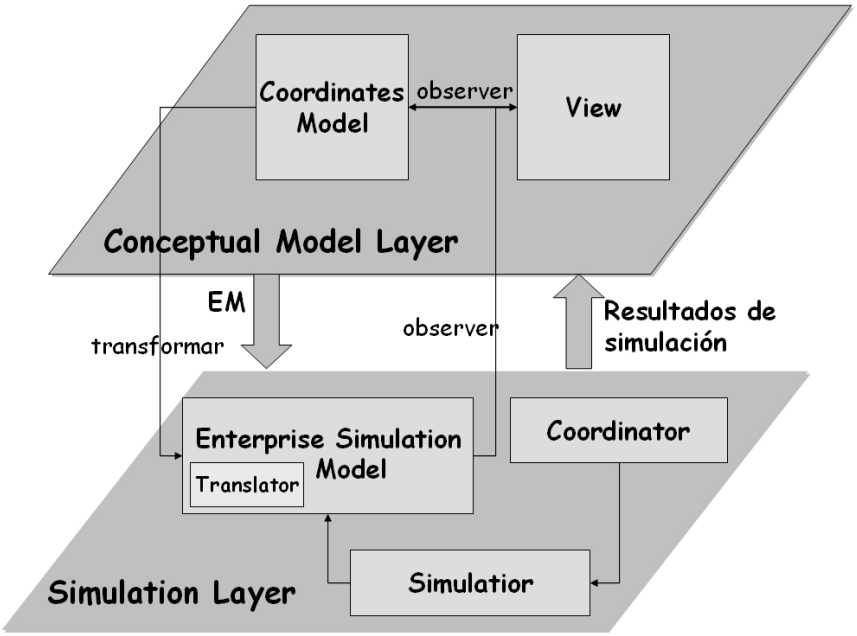


Figura 3.3: Arquitectura DE^{2}M

La capa de simulación tiene como funciones principales, obtener y ejecutar el modelo de simulación. A partir de la información que se tiene en el modelo conceptual, se provee funcionalidades para crear el modelo de simulación, ejecutarlo tanto localmente como en forma distribuida y obtener métricas de interés. Esta capa implementa el patrón de diseño MSVC (Model Simulation View Control) (Nutaro y Hammonds, 2004), el cual es una modificación al patrón MVC (Krasner y Pope, 1988). Este patrón de diseño, se enfoca sobre el desarrollo de simuladores basados en componentes, proponiendo una clara separación de conceptos entre *modelo*, *simulador* y *computación distribuida*.

Este paradigma de simulación basada en componentes, pone énfasis en la construcción de modelos de simulación en términos de componentes que interactúan, los cuales pueden ser reusados y/o sustituidos en el modelo de simulación e interpretados tanto aisladamente como en cooperación con otros componentes. Un componente, también conocido como *bloque de construcción* es definido por el BETADE-research group (Ver-

braeck y otros, 2002) como: *un bloque de construcción es una unidad auto-contenida, interoperable, reusable y reemplazable, encapsulando su estructura interna y proveyendo servicios o funcionalidades útiles a su entorno a través de una interfaz precisamente definida.*

Estos componentes pueden ser organizados en una jerarquía para formar un modelo de simulación, el cual a su vez puede ser considerado un componente para otro modelo. La motivación de emplear simulación basada en componentes, es la promesa de reducir costo y tiempo en el desarrollo del modelo, por la facilidad de reusar componentes existentes en lugar de crear modelos nuevos. Por otro lado, los componentes serían fáciles de mantener dado que cada uno de estos pueden ser testeados separadamente y aislados del resto de los componentes. Sin embargo, esto depende fuertemente de la definición de la interfaz. Esta característica es una de las diferencias entre ingeniería de software basada en componentes y la simulación basada en componentes: en simulación los componentes son altamente sensibles al contexto y muchas veces es difícil testear el componente aislado. Pero una diferencia mayor es que los componentes en simulación no solo intercambian datos en tiempo de ejecución sino que deben sincronizar sus tiempos locales de simulación.

La ventaja de emplear la organización de simuladores en base a componentes en esta propuesta es, como se dijo, que los componentes pueden formar un modelo de simulación cuando se los agrupa en una jerarquía, así este modelo de simulación es visto como el componente de mayor nivel en la jerarquía y puede ser ejecutado en un entorno de simulación. A su vez el componente de mayor nivel puede ser integrado con su entorno de ejecución y así formar un federado HLA. El mismo es también considerado un componente y de esta forma múltiples federados formarán una federación que es ejecutada en un entorno de ejecución (por ejemplo el RTI de HLA).

A continuación se describen las capas de la arquitectura y se presentan las etapas

propuestas para el desarrollo y evaluación de los modelos de empresa.

3.1.1. Capa de Modelado Conceptual

La construcción de un modelo de empresa (EM) es una tarea muy compleja, dado la cantidad de objetos que lo componen, la diversidad de los mismos y el elevado número de entidades intervinientes. Esto se ve agravado cuando la estructura de la empresa está distribuida geográficamente, donde sus miembros, pueden llegar a tener diferentes lenguajes y culturas, dificultando aún más la tarea de recolección de datos y de conceptualización de la organización.

Para atacar el problema de la complejidad, se emplea en esta capa un lenguaje de modelado de empresa que permite la construcción modular y jerárquica de los modelos y el empleo de diferentes vistas. Una vista permite obtener una perspectiva parcial de ciertos aspectos en los que se esté interesado, ocultando otros detalles.

El lenguaje seleccionado para la construcción de los EM es el lenguaje **Coordinates** (Mannarino y otros, 1999) el cual *es un lenguaje para representar el conocimiento de la empresa a través de diferentes dimensiones y que puede ser adaptado para reflejar las características propias de cada contexto empresarial*. Coordinates es un lenguaje soportado por el paradigma de orientación a objetos y presenta capacidades para representar una empresa a través de diferentes vistas: Vista del dominio, Vista de Tareas y Vista Dinámica. La *Vista del Dominio* abstrae los diferentes componentes de la organización junto con sus relaciones estáticas. Por ejemplo, en esta vista es posible definir un recurso *vehículo* para representar los medios de movilidad; más tarde este recurso puede ser especializado en recursos tales como *camión* y *auto*. De la misma manera es posible relacionar este recurso con otros recursos por ejemplo, el recurso *Vehículo* se puede relacionar con el recurso *Chofer*. La figura 3.4 muestra un ejemplo de esta vista.

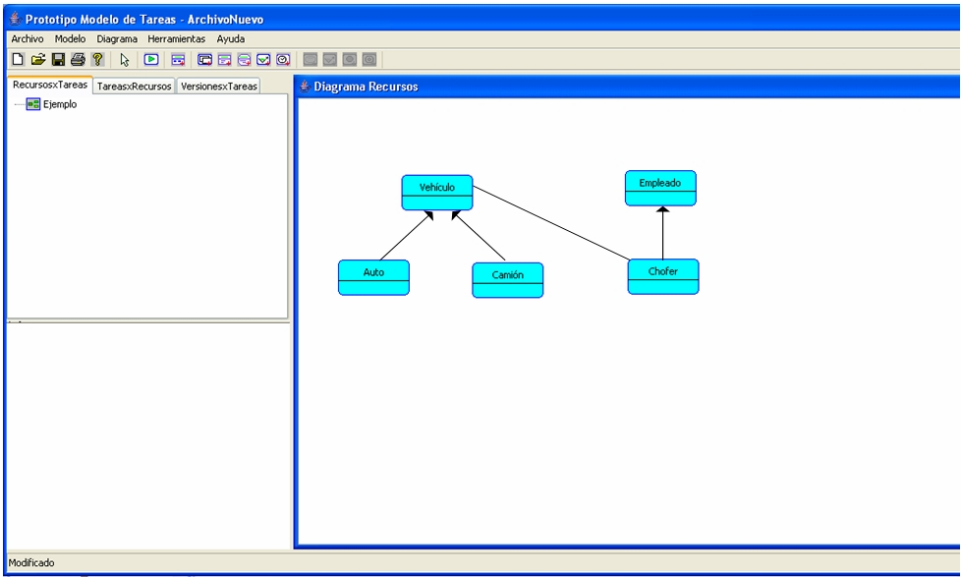


Figura 3.4: Vista del Dominio

La *vista de Tareas* identifica las actividades que se llevan a cabo y la manera en que pueden combinarse, implícita o explícitamente, para lograr sus objetivos. La figura 3.5 muestra un ejemplo de esta vista, donde las tareas: *Descargar Harina Industrial, Conformar OE* y *Completar OE* se relacionan a través de vínculos temporales (antes que e iguala a) y a través de los recursos que comparten.

Por último, la *Vista Dinámica* representa la evolución temporal de los recursos a través de su participación en las actividades de la organización, así como también la forma en que las actividades se encadenan. Un ejemplo de esta vista se muestra en la figura 3.6

Es de destacar que estas vistas no están aisladas unas de otras sino que integran un único modelo. Esta integración principalmente se logra al utilizar en las diferentes vistas el mismo vocabulario. El lenguaje es un lenguaje gráfico y se encuentra definido utilizando UML (Booch y otros, 1999).

Las componentes de un EM se definen en diferentes niveles de abstracción, y es posible identificar clases e instancias. De esta manera se pueden definir clases de acti-

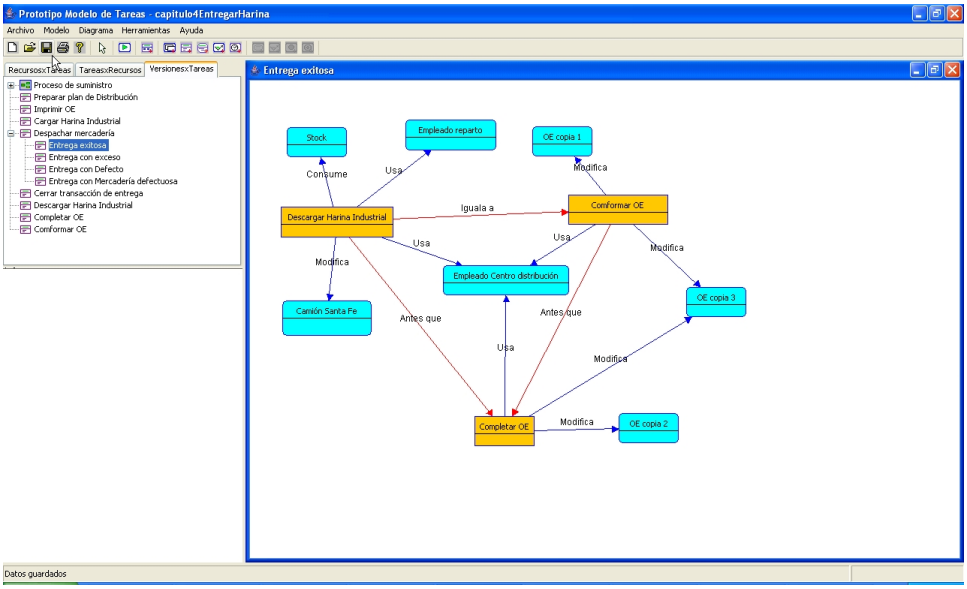


Figura 3.5: Vista de Tareas

vidades o clases de recursos como así también un recurso particular. La utilización de clases tiene como beneficio inmediato la posibilidad de reuso de los componentes, dado que la misma encapsula atributos y comportamiento.

La capa de de Modelado Conceptual, presenta funcionalidades para representar distintos diagramas que conforman las vistas definidas por el lenguaje. Es posible construir modelos de tareas en la Vista de Tareas, modelos de recursos en la Vista del Dominio y ciclos de vida de recursos en la Vista Dinámica. El **Modelo de tareas** describe una organización como un conjunto de **Tareas** que hacen uso de **Recursos** en función de alcanzar sus objetivos. Coordinates ofrece diferentes vínculos semánticos para vincular tareas con recursos, entre los cuales se pueden mencionar: usa, modifica, crea, procesa y elimina. Por ejemplo, una tarea *modifica* un recurso cuando el recurso cambia de estado debido a su participación en la ejecución de la misma. La relación *usa* representa que el recurso es visto como una herramienta durante la ejecución de la misma. Este vínculo tiene una semántica muy débil para indicar el rol del recurso con respecto a

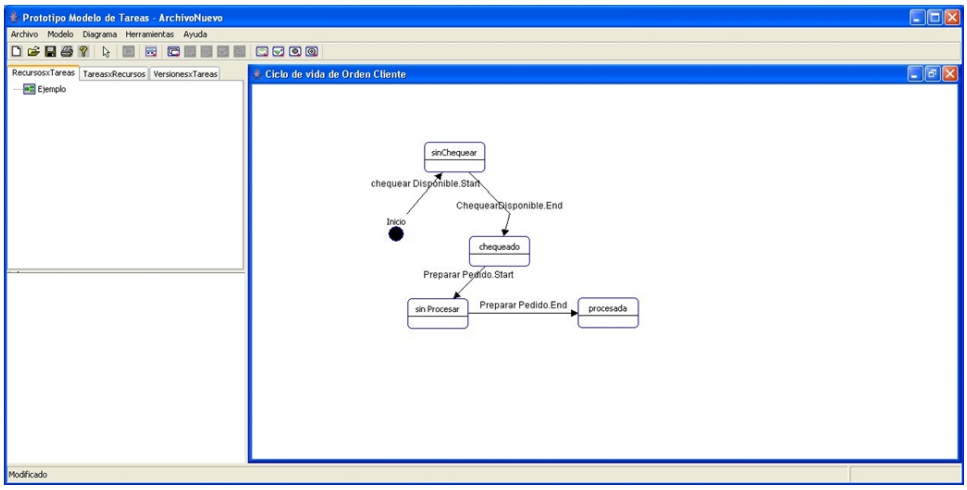


Figura 3.6: Vista Dinámica

una tarea, el recurso puede o no cambiar su estado durante la ejecución. Las relaciones *crea / elimina* indican que el recurso es generado o eliminado en el dominio cuando la tarea se ejecuta. También las tareas están vinculadas entre sí a través de relaciones temporales (Allen, 1984) que indican el orden en que las mismas pueden ser ejecutadas: *antes que, encuentra, solapa, iguala, termina, durante y comienza*. Coordinates supone la descripción de Tareas en diferentes niveles de detalle. Un *Proceso* debe ser la raíz de una jerarquía de descomposición y una *Tarea Compuesta* es un nodo intermedio del árbol de descomposición. Tanto una *Tarea Compuesta* como un *Proceso* se describen a través de una o más *Versiones de tarea* cada una, representando una forma particular de realizarla. Así una versión de tarea puede diferenciarse de otra por la combinación diferente de recursos que usa, las subtareas en que se descompone, las condiciones que deben cumplirse para que dicha versión se ejecute, etc. Una *Versión de Tarea* se describe en términos de *tareas, recursos, relaciones temporales* y relaciones entre *Tareas* y *recursos*. El lenguaje permite formular precondiciones que restringen la ejecución de la tarea, éstas se expresan como los estados de los recursos que debe encontrar la tarea en el momento de ejecutar. El **Modelo de Recursos** se enfoca sobre las característi-

cas estructurales de los recursos. Relaciones tales como *especialización, composición* y *asociación* son usadas para relacionar los diferentes recursos. El **ciclo de vida del recurso** describe la manera en que un recurso evoluciona (cambia de estado) cuando participa en las tareas. Este modelo se basa en los *diagramas de transición de estados* propuestos por Harel (Harel, 1987),(Harel y Gery, 1996) llamados *Statechart*. Así, la evolución del recurso se representa como *estados* y *transiciones* entre ellos. Estas últimas son causadas por la ejecución de tareas. En estos diagramas se puede analizar cómo el recurso cambia de estado por la participación en las tareas, pero también es posible identificar cuáles son los estados del recurso que son precondiciones para que una tarea se ejecute. Por ejemplo si se tiene una transición del estado **A** a otro estado **B** rotulada con la tarea **T**, esto puede leerse como: El recurso pasa del estado **A** al estado **B** cuando la tarea **T** es ejecutada, y es condición necesaria que el recurso esté en el estado **A** para que la tarea **T** pueda comenzar a ejecutarse.

El objetivo de la capa de modelado conceptual es brindar los servicios necesarios para construir un modelo de empresa y sus procesos. Siguiendo los lineamientos propuesto por el lenguaje de modelado adoptado, éste propone la construcción del EM desde una perspectiva funcional. Así un dominio de empresa se describe como un conjunto de tareas que se vinculan temporalmente y que interactúan a través de un conjunto de recursos que comparten. Los modelos de tareas son la vista funcional de la organización sobre la que está basada el desarrollo del EM. La construcción del modelo, está sustentada por las funciones que brinda la arquitectura, permitiendo la formulación incremental e iterativa de los modelos. La interface gráfica representada por la componente *View* de esta capa (figura 3.3), facilita la tarea de construir el EM, dado que los diferentes conceptos tienen su representación gráfica asociada.

El capítulo 4 describe con detalle el diseño de esta capa, mientras en este apartado solo será presentada una introducción de los principales conceptos del lenguaje

Coordinates.

3.1.2. Capa de Simulación

La capa de simulación (figura 3.3) está diseñada para prestar servicios relacionados a la transformación del modelo conceptual en un modelo de simulación (SM), ejecutar el modelo de simulación, (tanto localmente como en forma distribuida) y la obtención de métricas de interés que sirvan de base para analizar los procesos simulados usando datos cuantitativos. Se utiliza el formalismo DEVS para la representación del modelo de simulación. Este formalismo, como se indicó en el capítulo 2, permite la construcción de los modelos de simulación en forma jerárquica y modular facilitando la representación de modelos complejos.

En esta tesis, se utilizó el paradigmas de simulación basado en componentes, donde los componentes o bloques de construcción del modelo de simulación para el dominio específico de los procesos de empresa se definen formalmente. Es de destacar que esta disciplina está en desarrollo y constituye uno de los campos de investigación de mayor interés en la actualidad (Barros y otros, 2004). En la definición de los bloques de construcción se siguieron las recomendaciones presentadas en Verbraeck y Valentin (2008) de manera de cumplir con los objetivos de *reusar* y *reemplazar* que un bloque de construcción debe tener. Así un bloque de construcción o componente, debe ser:

autocontenido expresando la idea que el estado que un componente tiene en un dado instante es el resultado del valor de sus atributos como también de sus funciones o procesos, estos últimos son representados a través de elementos que un componente tiene;

interoperable en el sentido de tener la habilidad de intercambiar información y de usar aquella que ha sido intercambiada (IEEE, 1990), para que este intercambio

sea posible es necesario que los componentes conozcan los otros componentes existen en el sistema, con los cuales podría intercambiar información;

encapsular su estructura interna indicando que tanto sus elementos, como su estructura interna, deben estar ocultos a los otros componentes, detrás de su interfaz. Esta interfaz, oculta completamente el código en el interior y elimina la posibilidad que un diseñador destruya la lógica del componente o de algunos de sus elementos;

proveer servicios y funciones útiles los cuales están representados por sus elementos. De esta forma, un componente puede tener varios elementos para los diferentes servicios o funcionalidades que éste provee;

interfaz bien definida que le permitan a un usuario del componente entender en todo momento los posibles usos del mismo, ya sea durante la ejecución de la simulación, el análisis de resultados como también durante el desarrollo del modelo. La interfaz es la visualización del estado del bloque de construcción incluyendo su operación actual durante la ejecución de la simulación. Esta interfaz puede tener parámetros que le permiten al desarrollador del modelo establecer el estado inicial, distribuciones estadísticas, y otros parámetros de las funciones de un bloque de construcción;

Los bloques de construcción definidos en esta capa, están relacionados con los conceptos utilizados en el modelo conceptual, encapsulando su comportamiento dinámico. Cada bloque de construcción es representado por un modelo DEVS que define su comportamiento interno a través de las funciones de transición de estados (delext y delint) mientras que su interfaz queda definida por los puertos de entrada y salida y los valores posibles para dichos puertos. Estos modelos se pueden asociar a través de acoplamientos

entre sus puertos, dando como resultado componentes de mayor nivel con una estructura y comportamiento particular. Además, el hecho de separar modelo de su simulador, permite reusar el modelo de simulación para entornos locales o distribuidos. Así el SM es generado una vez y ejecutado en ambos entornos, asociando para ello máquinas de simulación diferentes.

El uso de componentes en esta capa está estrechamente relacionada a la filosofía de construcción del modelo conceptual en la capa superior. Así, cada concepto en la capa superior tiene asociado su modelo DEVS (correspondiente a un bloque de construcción) que representa su comportamiento. Cuando un concepto participa de un modelo conceptual, una instancia del modelo DEVS es generada e incorporada al modelo de simulación. Por ejemplo, el concepto *Tarea* en el modelo conceptual representa una actividad llevada a cabo en la empresa. Este concepto tiene asociado su modelo de comportamiento representado por un modelo DEVS *TareaDevs*. Cuando una tarea particular es generada, tal como *generar orden de pedido*, una instancia de *TareaDevs* es generada con los elementos necesarios para representar el comportamiento del concepto en el dominio. Luego esta instancia participará del modelo de simulación que será ejecutado.

Un elemento importante en esta capa es el *Traductor*(Translator), el cual es el encargado de realizar la transformación entre el EM al SM siguiendo las reglas definidas y presentadas en el capítulo 5. Cuando la acción de simular el EM se ejecuta, la capa superior delega en la capa de simulación esta tarea.

La figura 3.7 muestra un esquema de una simulación local usando la arquitectura. A partir de las definiciones realizadas en el modelo conceptual de empresa, la capa de simulación se encarga de generar el modelo de simulación, ejecutarlo usando una máquina de simulación capaz de interpretar el modelo en el entorno local (en este caso llamado *Coordinator*) y mostrar los resultados de la simulación. Luego, a partir de esta

información el usuario podrá introducir mejoras en el modelo conceptual basándose en datos cuantitativos obtenidos del proceso de simulación. En el ejemplo mostrado en la figura 3.7 los gráficos de barra muestran los tiempos de ejecución de las tareas y el nivel de stock de los recursos que participaron de la simulación.

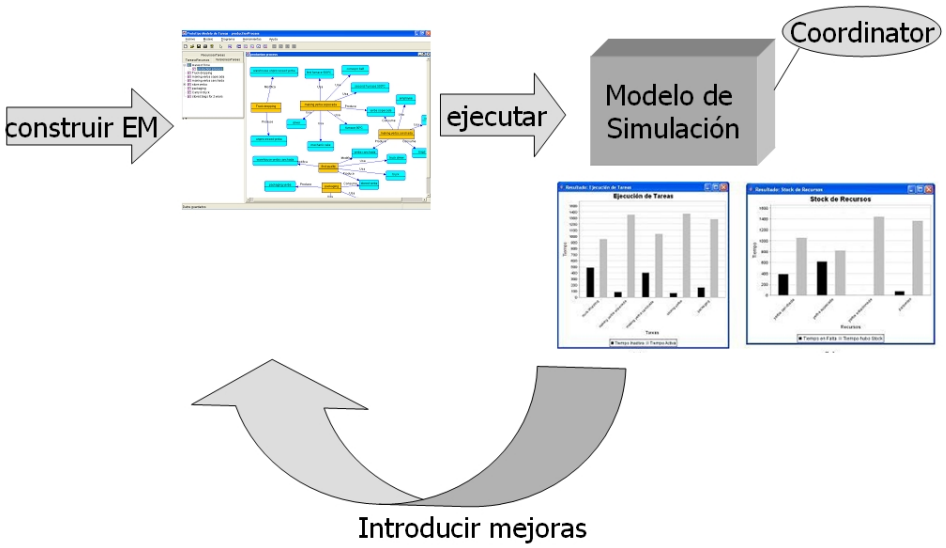


Figura 3.7: Esquema de simulación local

La figura 3.8 muestra el esquema de una simulación distribuida. De igual forma que en simulación local, el modelo de simulación es generado a partir del modelo conceptual definido en la capa superior. Luego el modelo de simulación forma parte de un federado capaz de interactuar con otros en una federación bajo HLA, esto quiere decir que el federado que se genera cumple con la especificación del estándar utilizado. Este federado, llamado *federateE2M*, tiene asociado una máquina de simulación que permite la ejecución distribuida -en el gráfico CoordinatorE2M- El coordinador (CoordinatorE2M) altera la máquina de simulación DEVS con el objetivo de permitir que el modelo de simulación DEVS ejecute en un entorno distribuido bajo HLA.

Por otro lado, cuando los federados interactúan en una federación, es necesario que definan las interacciones y objetos que pueden intercambiar entre ellos. Estos datos

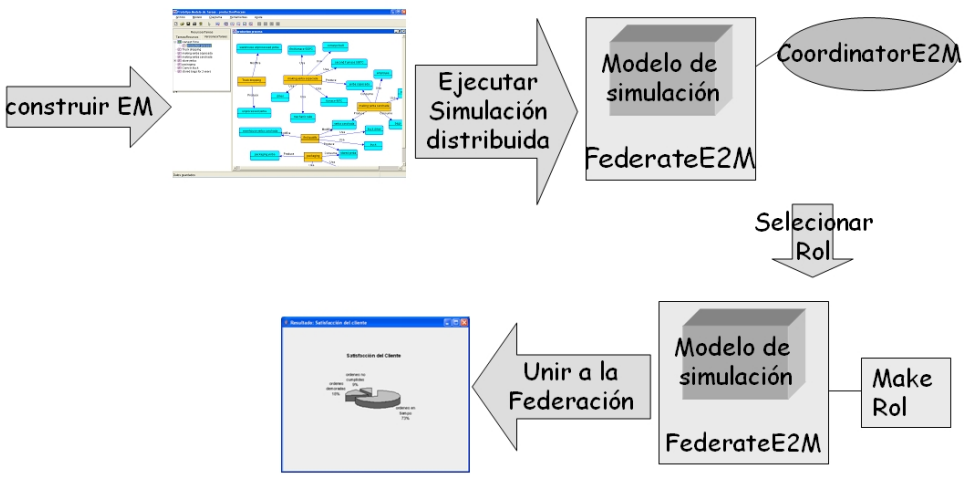


Figura 3.8: Esquema de simulación Distribuida

se encuentran en el *FOM*, (como se describió en el punto 2.2.2) entonces cuando un federado se une a una federación, debe informar qué datos envía y cuáles puede recibir, a la vez que se comprometen a interpretar en forma correcta los datos recibidos. Esto es conocido como la *interoperabilidad semántica*, es decir, no solo es suficiente poder intercambiar datos, sino que es necesario interpretarlos correctamente. Para lograr esta interoperabilidad en esta propuesta se utilizaron roles que los federados pueden jugar, estos roles definen el conjunto de datos que un federado puede enviar y recibir y está basado sobre un FOM definido de acuerdo a los objetivos que se plantean en la federación. El formato FOM provee de interoperabilida sintáctica, cada federado se compromete a interpretar correctamente estos datos intercambiados.

El empleo de roles, permite que un federado puede participar de diferentes federaciones jugando diferentes roles. En esta propuesta solo se definieron roles en el contexto de cadenas de suministro, dejando como trabajos futuros la definición de otros roles en otros dominios de empresa.

3.1.3. Pasos en la Construcción de un Modelo DE²M

En esta sección se presentan los pasos a seguir para construir un modelo DE²M, a través del uso de la arquitectura descripta. La figura 3.9 muestra el diagrama de actividades que sintetiza las acciones ejecutadas para la construcción de modelos.

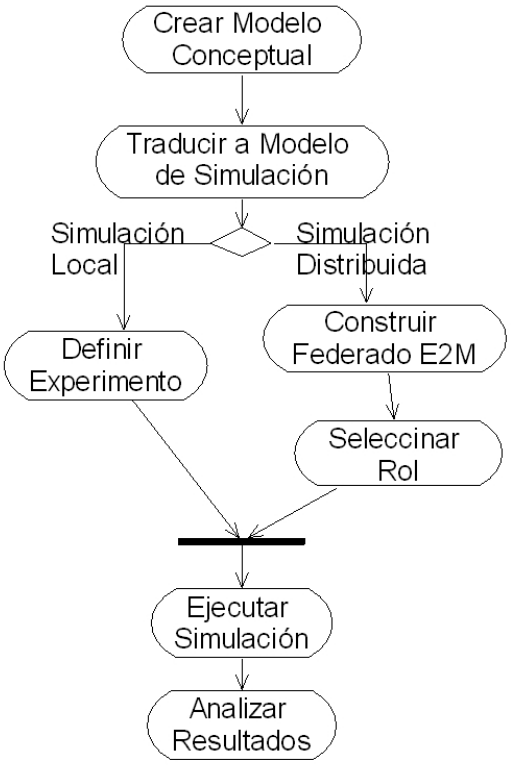


Figura 3.9: Metodología de desarrollo de un DE²M

Inicialmente, el desarrollador construye el modelo conceptual de la empresa conocido como el EM. El EM está conformado por los modelos de tareas, de recursos y el ciclo de vida de cada uno de los recursos definidos. Estos modelos representan las diferentes vistas del EM. Este paso se encuentra detallado en el capítulo 4. Una vez finalizado el mismo, el desarrollador podrá ejecutarlo con el objetivo de analizar su comportamiento dinámico. En este momento, el modelo de simulación (SM) es generado por el componente de la arquitectura denominado *Translator*. El traductor, *Translator*, es el

encargado de tomar el EM y transformarlo en un SM equivalente mediante la aplicación de un conjunto de reglas de traducción de modelos (se describen en detalla en el capítulo 5). Este SM puede ser ejecutado, es por ello que se lo llama E^2M o modelo de empresa ejecutable (de su sigla en inglés Executable Enterprise Model). Seguidamente, se debe tomar la decisión si el E^2M se ejecutará localmente o en forma distribuida, esta elección se hará teniendo en cuenta los objetivos de análisis.

Así, si se selecciona simulación local, sólo los procesos internos de la organización que aparezcan definidos en el EM, serán ejecutados y evaluados. Si estos procesos corresponden a una empresa, entonces se estará en presencia del análisis de los procesos internos de la misma. Si por el contrario, se decide ejecutar una simulación distribuida, lo que se estará analizando es la relación de estos procesos internos como parte de procesos más generales inter-empresas.

Para poder realizar esta simulación, el SM es generado, el cual formará parte de un federado que es considerado parte de un sistema mayor, compuesto por un conjunto de federados que se pueden ejecutar juntos, formando una federación. Una vez que este paso se concluye, si la simulación seleccionada es distribuida, antes de ejecutar definitivamente la simulación, es necesario seleccionar el rol que el federado juega en la federación. Este rol, le permite al federado definir el conjunto de interacciones que está interesado en recibir desde otros federados como también de aquellas interacciones que enviará.

Una vez ejecutado el EM, ya sea local o distribuido, la información generada puede ser expresada mediante métricas que son visualizadas en la interface gráfica de manera que se puedan tomar decisiones sobre los procesos de la empresa analizando su comportamiento. A partir de esta información es posible modificar el modelo conceptual para analizar diferentes alternativas, como puede ser por ejemplo agregando o eliminando recursos, cambiando el orden de ejecución de las tareas, agregando o eliminando ta-

reas, agregando nuevas versiones para una tarea dada, etc. Así los diferentes escenario pueden ser evaluados cuantitativamente detectándose problemas y analizando posibles soluciones que pueden tener una implementación en el mundo real de la empresa con resultados conocidos.

3.1.4. Conclusiones

En este capítulo si hizo una introducción a la propuesta realizada en esta tesis. La arquitectura en capas presentada, define el contexto adecuado para la construcción de un modelo conceptual de la organización y su ejecución con el objetivo de hacer una análisis de su comportamiento dinámico. Las posibles formas de ejecución planteadas, local y distribuida, se focalizan sobre dos dimensiones diferentes de análisis de los procesos: interna y externa. La arquitectura en capas presenta el proceso de simulación como un servicio, otorgando beneficios en cuanto al tiempo de obtención del modelo de simulación, la validez del mismo y la interpretación de los resultados.

4

Modelo Conceptual de Empresa

En este capítulo se aborda la descripción detallada de los conceptos y características usadas en la arquitectura para brindar soporte al desarrollo del modelo conceptual de la organización. Se describe principalmente la capa de modelado conceptual (Conceptual Model Layer) utilizada en la arquitectura para el desarrollo de los modelos de empresa. Se dará a conocer el diseño de esta capa, describiendo las principales funcionalidades de la misma.

Uno de los objetivos principales de la arquitectura DE^{2}M es apoyar el proceso de modelado de empresas, siendo éste el punto de partida para describir, evaluar, analizar, mejorar, reorganizar y ordenar Organizaciones. El modelo de Empresa (EM), encapsula las dimensiones de procesos, de información, de organización, de materiales, de costos y dinámicas. En la construcción de estos modelos, se ha adoptado el lenguaje Coordinates el cual ha demostrado cumplir con los requerimientos de un lenguaje de modelado de empresas. Este lenguaje está basado sobre tres principales vistas: *Vista del Dominio*, *Vista de Tareas* y *Vista Dinámica* las cuales cubren los aspectos estáticos, funcionales y dinámicos de una organización. La *Vista de Tareas* identifica los procesos que hacen uso de diferentes Recursos para alcanzar las metas de la organización. Las características estructurales de las diferentes entidades de una organización son encapsuladas en la

Vista del Dominio. Finalmente, la *Vista Dinámica* enfoca sobre la evolución temporal de los Recursos a través de su ciclo de vida como una consecuencia de participar en las diferentes Tarea.

Con el fin de lograr coherencia en la construcción de los modelos, se consideraron los conceptos adoptados por el lenguaje de modelado seleccionado. Principalmente, Coordinates considera el término Empresa como *un conjunto de tareas que combinan de alguna manera los recursos disponibles para lograr sus objetivos*. Consecuentemente, dicho lenguaje se focaliza sobre la vista funcional de la organización, centrando el desarrollo de los modelos en el diseño conceptual de los denominados *Modelos de Tareas*: un dominio de empresa se describe funcionalmente a través de un conjunto de *Tareas* que se vinculan *temporalmente* e interactúan a través de los *Recursos* que comparten.

La capa de modelado conceptual (Conceptual model Layer) de la arquitectura propuesta, se diseñó de manera de apoyar el proceso de modelado de empresa, utilizando para ello el lenguaje Coordinates. En consecuencia, esta capa aborda la construcción del modelo de empresa a través de las diferentes vistas. La figura 4.1 esquematiza un modelo de empresa representado por las diferentes dimensiones, esto es, el modelo de empresa es visto como un único elemento formado por distintas vistas que se complementan para dar sentido a esa unidad.

Con el fin de dar soporte a la construcción de estos modelos de empresa, desde la perspectiva del lenguaje seleccionado, se definieron los siguientes requerimientos:

- construir el EM usando un proceso iterativo
- utilizar un único lenguaje de modelado
- concebir un modelo de empresa único, descripto a través de diferentes dimensiones
- permitir futuras especializaciones del modelo

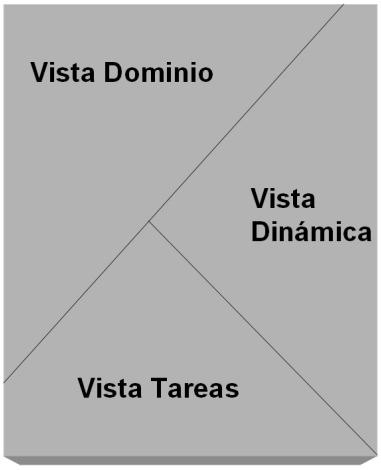


Figura 4.1: Vistas de Coordinates

- permitir el reuso de modelos parciales

En la siguiente sección se aborda el diseño de la capa de modelado conceptual (Conceptual Model Layer) que da soporte al proceso de modelado de empresa. Se analiza en su desarrollo, el cumplimiento de los requerimientos planteados con anterioridad y se hace una descripción detallada de sus componentes justificando las decisiones adoptadas para cumplir con las especificaciones del lenguaje como así también aquellas necesarias para facilitar el proceso de desarrollo de los modelos.

4.1. Diseño de la Capa de Modelado Conceptual

El modelado de empresas es una actividad inherentemente compleja, no solo por la cantidad de objetos que constituyen el modelo de empresa (en términos del lenguaje usado, estos componentes serán las tarea, los recursos, las restricciones, los estados, las transiciones, etc) sino también por la cantidad de entidades que intervienen en este proceso (como ser las diferentes unidades organizacionales, las ramas de una organización, los departamentos, las áreas que se encuentran involucradas).

Consecuentemente, para hacer frente a la complejidad se requiere no solo de un lenguaje suficientemente expresivo que permita describir sin ambigüedad los diferentes aspectos de una organización, sino también es necesario adoptar una visión modular del problema complejo, haciendo uso de técnicas de descomposición modular en el proceso de modelado, lección bien aprendida de la disciplina de ingeniería del software. Así, un problema complejo es descompuesto en problemas más simples para su tratamiento de manera que pueda ser abordado por una persona o grupo de personas competentes en el tema.

El lenguaje adoptado en esta tesis, ha demostrado ser suficientemente expresivo para la representación de un dominio de empresa. Las características del mismo han sido tenidas en cuenta en el diseño de la capa de modelado conceptual. Principalmente, el lenguaje posibilita la definición de los componentes de un EM en diferentes niveles de abstracción, combinando clases e instancias. En este sentido, el lenguaje permite definir clases de modelos, que representan características y comportamientos generales de la empresa y abstraen entre otros aspectos, los tipos de procesos que se llevan a cabo, los tipos de recursos que se disponen, así como también los recursos particulares con los que cuenta la empresa en un determinado momento. Este lenguaje hace pocas suposiciones del dominio que representa (mínimo compromiso ontológico)((Gruber, 1993)) en cuanto a que se define a través del conjunto de objetos esenciales para representar a un contexto empresarial, dejando abierta la posibilidad de extensión.

Teniendo en cuenta las características del lenguaje y los requerimientos funcionales planteados para la arquitectura, la capa de modelado conceptual se diseñó de acuerdo al estilo arquitectónico Document-View (Buschmann y otros, 1996) como se observa en la Figura 4.2. El estilo arquitectónico seleccionado, propone separar el modelo de su interfaz gráfica permitiendo manipular conceptos que pueden adoptar diferentes formas gráficas (Gutierrez y otros, 2001b). Los componentes que conforman la capa de

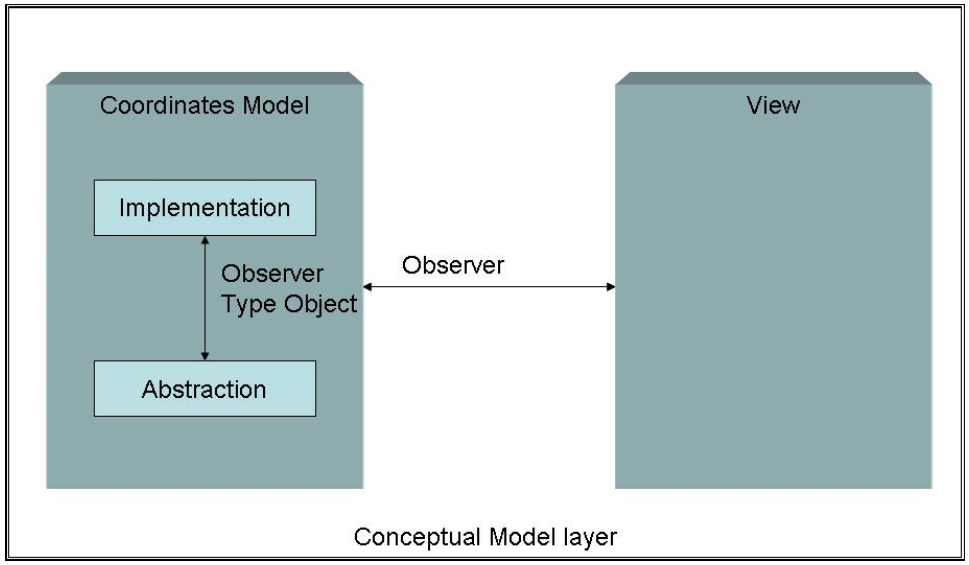


Figura 4.2: Arquitectura de la capa de modelado conceptual

modelado conceptual son dos:

Coordinates Model encapsula la información de una organización de producción o de servicio. El componente *Abstraction* contiene las Tareas y Recursos que son instanciados en el componente *Implementation*. *Coordinates Model* abarca las tres vistas definidas en Coordinates.

View constituye la interfaz gráfica a través de un conjunto de entidades gráficas.

El componente *Coordinates Model*, tiene a su vez dos componentes: *Abstraction* e *Implementation* reflejando los conceptos de Clase-Instancia proporcionados en el lenguaje Coordinates. En el componente *Abstraction* se definen los elementos conceptuales que forman el vocabulario del dominio de modelado de empresa, referido como el *dominio de abstracción*, con los cuales pueden construirse los modelos de Tareas y Modelos de recursos mientras que en el componente *Implementation* se usan dichos elementos, es decir, este componente contiene instancias u ocurrencias de los conceptos definidos en *Abstraction*. Cada componente tiene su vista gráfica asociada que se encuentra definida

en el componente *View*. De esta manera, tanto los componentes de los modelos como las estructuras básicas empleadas para crearlos tienen una representación gráfica. Entre los elementos de *Abstraction* y los de *Implementation* se tiene una relación Clase-Instancia, (patrón Type -Object (Johnson y Wolf, 1998)) esto es, cada elemento de *Abstraction* abstrae la noción de una clase cuyas instancias son los elementos de *Implementation*. Una implementación representa el uso de un elemento conceptual específico en un contexto particular. El patrón *Observer* (Gamma y otros, 1996) mantiene consistencia entre las componentes (i) *Abstraction* e *Implementation* y entre (ii) *Coordinates Model* y *View*.

Coordinates Model, integra las tres vistas de modelado propuesta por el lenguaje Coordinates (Gutierrez y otros, 2001a). Así, los servicios provistos reflejan las acciones posibles en cada vista. La **Vista del Dominio** representa los conceptos generales con los cuales se crearán los *modelos de recursos*. Acciones tales como crear diagrama de recurso, definir un recurso, relacionar recursos, son algunas de las posibles acciones en esta vista. La **Vista de Tareas**, estará representada por el *Modelo de tareas*, en la construcción de estos modelos interviene el concepto de instancia. Los modelos de tareas abarcan la descomposición de tareas en subtareas, las relaciones entre tareas y recursos y las relaciones temporales entre tareas. Las posibles acciones a realizar en esta vista son: crear instancias de tareas, crear instancias de recursos, crear instancias de relaciones tareas-recursos, crear relaciones temporales entre tareas, crear una versión de tarea, agregar o eliminar instancias a la versión de tareas y crear tareas. Finalmente, la **Vista Dinámica** está representada a través de diagramas de transición de estados. En esta vista se modela el *ciclo de vida del recurso*, el cual refleja los posibles estados que el mismo adopta durante su vida al participar de las diferentes tareas. Las acciones posibles en esta vista son: crear un estado de un recurso, eliminar un estado, crear / eliminar una transición, asociar una transición a la ejecución de una tarea y crear una

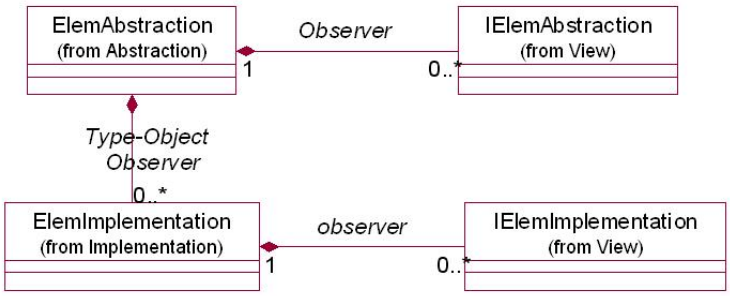


Figura 4.3: Relación entre los Elementos de Coordinates Model y View

restricción a una transición. Los recursos que fueron definidos en la Vista del Dominio, deben tener asociado un ciclo de vida que indique la evolución del mismo al participar de las diferentes tareas. De esta forma, los estados de los recursos pueden ser considerados como condiciones para la ejecución de tareas, y las tareas los eventos que causan la transición entre los diferentes estados del ciclo de vida del recurso.

View implementa la interfaz gráfica de los elementos definidos en *Coordinates Model*. Cada elemento que está definido en *Coordinates Model* tiene su representación gráfica asociada. Esta interfaz gráfica también comprende la definición de los eventos a los cuales el elemento puede responder. Por ejemplo estos eventos pueden ser: ocultar un elemento gráfico, mostrar un elemento gráfico, cambiar posición del elemento gráfico. Pero también existen otros eventos que son más específicos del modelo al cual el gráfico representa, como puede ser por ejemplo, crear una instancia de ese elemento, agregar un diagrama, descomponer el elemento, etc. La figura 4.3 muestra la relación semántica entre los elementos de *Coordinates Model* y *View*, donde la clase *ElemAbstraction* representa los elementos pertenecientes a *Abstraction*, la clase *ElemImplementation* representa los conceptos de instancias. Por último las clases *IElemAbstraction* e *IElemImplementation* representan las interfaces gráficas asociadas a los elementos de abstracción e implementación respectivamente.

La capa de modelado conceptual, al dar soporte a la construcción de los modelos

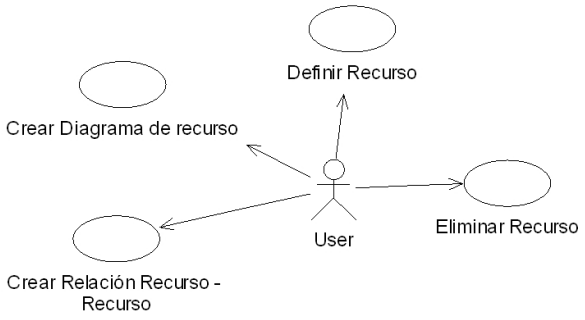


Figura 4.4: Diagrama de Casos de Uso de la Vista del Dominio

conceptuales de la organización, debe proveer funcionalidades para crear las diferentes vistas que conforman el EM según el lenguaje adoptado: Vista del Domino, Vista de Tareas y Vista dinámica. Estas funcionalidades fueron divididas de acuerdo a cada vista, asociando un conjunto de posibles acciones a realizar en cada una de ellas. Se presenta a continuación el detalle de cada vista indicando no solo las funcionalidades asociadas sino también los elementos necesarios para dar soporte a las mismas.

4.1.1. Vista del Dominio

Esta vista abarca principalmente los recursos y sus relaciones definidos en el componente *Abstraction*. A través de esta vista es posible definir la estructura de los recursos representando la manera en que los mismos dan soporte a los servicios ofrecidos por las tareas. La figura 4.4 muestra el diagrama de casos de uso de esta vista, donde se identifican las principales funcionalidades que la misma provee. Así, se permiten crear/eliminar un recurso, crear un diagrama de recursos, crear/eliminar relaciones entre los recursos.

Para poder llevar a cabo estas funcionalidades, se definieron los elementos del lenguaje que participan de esta vista. El diagrama de clases de la figura 4.5 muestra estos

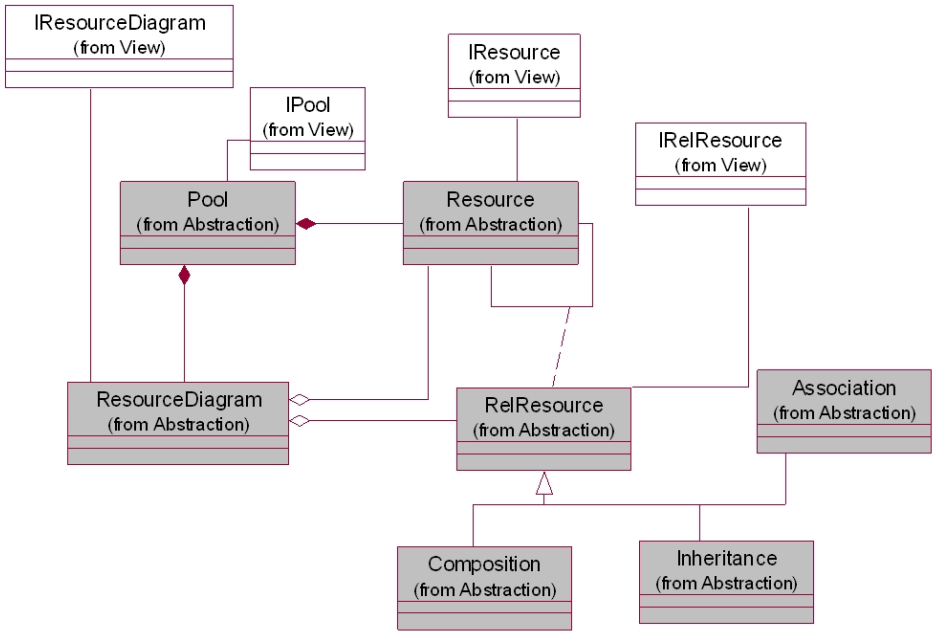


Figura 4.5: Diagrama de Clases de la vista del Dominio

elementos. Cuando un recurso es definido, éste estará incluido en un repositorio (clase *Pool*) que representa el reservorio o contenedor de todas las definiciones realizadas, así puede verse a esta clase como un diccionario de términos a ser utilizados en la construcción de los diagramas. Cuando un diagrama de recursos se crea, éste contendrá un conjunto de recursos y relaciones entre estos, que definen la estructura de los mismos. Los diagramas de recursos definidos, también formarán parte del repositorio y pueden ser vistos como las definiciones de estructuras y relaciones entre los conceptos que participan del mismo. La clase *ResourceDiagram* representa un diagrama de recursos, en el cual los recursos participantes están relacionados a otros recursos a través de diferentes relaciones. La clase *Resource* representa el concepto Recurso definido en el lenguaje Coordinates. Estos, al ser definidos forman parte del repositorio y pueden pertenecer a un diagrama de recursos, en el cual aparecerán relacionados con otros recursos. Por último, la clase *RelResource* representa las relaciones entre los recursos. Esta clase es

una clase abstracta cuyas especializaciones son: *Inheritance*, *Composition*, *Association*, representando las relaciones de especialización, composición y asociación respectivamente. La relación de especialización (clase *Inheritance*) simboliza una relación *es-un* entre dos recursos. Es utilizada para indicar que un recurso es una definición más específica de otra dada, por ejemplo para indicar que un recurso *remito* es un tipo de *documento comercial*. La relación de composición (clase *Composition*) tiene la semántica *parte-todo*. Esta relación es utilizada cuando un recurso está compuesto por otros recursos. Por ejemplo, se puede representar un *remito* formado por tres copias: *original*, *duplicado* y *triplicado*. Finalmente la relación de asociación (clase *Association*) identifica una relación débil entre dos recursos. Por ejemplo se puede establecer una relación entre el recurso *empleado administrativo* que se encarga de elaborar el remito y el recurso *remito*.

Estos elementos pertenecen al componente *Abstraction* y cada uno tiene su interfaz gráfica asociada, las cuales aparecen en el diagrama de clases de la figura 4.5 como las clases no coloreadas y son: *IPool, IResource, IResourceDiagram, IRelResource*. Las interfaces gráficas pertenecen al componente *View* definido en la arquitectura de esta capa.

Para poder entender como son generados los diagramas en esta vista se analizan los casos de uso definidos anteriormente.

Caso de uso: Definir un Recurso. Definir un recurso significa crear un nuevo concepto o entidad de modelado dentro de la vista del Dominio. Cuando el recurso se define debe tener un nombre asociado el cual forma parte del vocabulario de modelado. El recurso que se define es una instancia de la clase *Resource*.

La creación de un nuevo recurso, implica la generación de un evento que la interfaz gráfica puede entender y procesar como tal, actuando en consecuencia. Es posible que

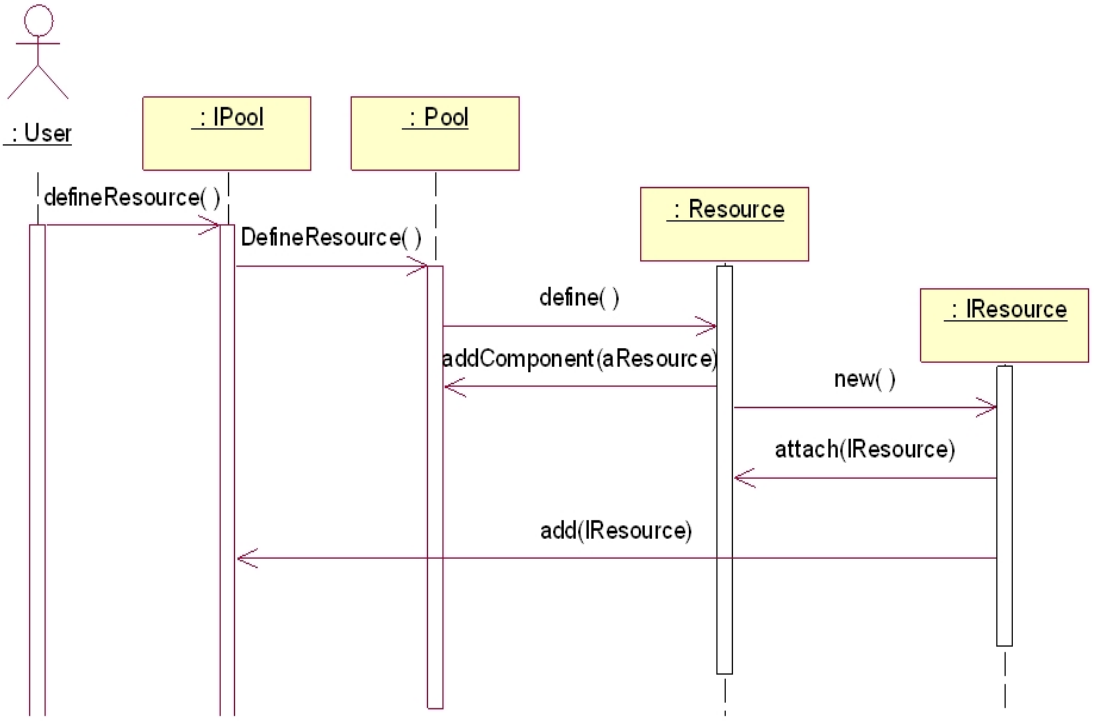


Figura 4.6: Diagrama de secuencias correspondiente a Definir un Recurso

este evento sea enviado tanto a la interfaz gráfica del repositorio *IPool* como a la interfaz gráfica del diagrama de recursos *IResourceDiagram*. En el primer caso, el nuevo recurso no formará parte de un diagrama, sino que el mismo es definido para su posterior uso en alguno de los modelos que conforman el EM. En el segundo caso, en cambio, el nuevo recurso forma parte de un diagrama de Recursos, donde se pueden definir relaciones con otros recursos que participen de ese diagrama. En ambos casos, sin embargo, el recurso que se crea forma parte del repositorio *Pool* como un concepto más de modelado. El diagrama de interacción de la Figura 4.6 muestra el caso en que *IPool* recibe el evento para la creación de un Recurso. Cuando la clase *Resource* recibe el mensaje *Define*() crea su interfaz gráfica y se agrega como elemento del repositorio (*Pool*).

Caso de uso: Crear un Diagrama de Recursos. Esta acción tiene como finalidad crear una instancia de la clase *ResourceDiagram*, es decir un diagrama de recursos inicialmente vacío. Una vez creado el diagrama, es posible incorporar recursos ya definidos, crear nuevos recursos que participan del diagrama y crear relaciones entre los recursos involucrados en el diagrama. El diagrama de secuencia de la figura 4.7 muestra las interacciones que se llevan a cabo para crear el diagrama de recursos que forma parte de la vista del Dominio. Como se puede apreciar, cuando se dispara el evento *newResourceDiagram*, se crea una instancia de la clase *ResourceDiagram*, el cual crea su interfaz gráfica. Dado que existe un *observer* entre *ResourceDiagram* y su interfaz gráfica *IResourceDiagram*, una vez que se crea *IResourceDiagram*, éste se agrega como observador (método *attach*()) de *ResourceDiagram*. Luego, siguiendo las interacciones mostradas en la figura 4.7, es posible incorporar nuevos recursos al diagrama (método *newResource()*) o crear nuevas relaciones entre los elementos del diagrama (método *newInheritance()*). Es de destacar que otros tipos de relaciones pueden ser generadas en el diagrama. Solo se muestra a modo de ejemplo el envío del mensaje *newInheritance* que representa la creación de una relación de especialización. La creación de un diagrama de recursos es una de las principales funcionalidades de la Vista del Dominio, desde donde es posible identificar los recursos y sus relaciones en un dominio de empresa. La creación de estos diagramas en la vista del Dominio, permite no sólo estructurar los recursos ya definidos, indicando composición, asociación y especialización, sino también es posible definir nuevos recursos que participarán de estos diagramas.

Caso de Uso: Eliminar un Recurso. La eliminación de un recurso de esta vista, implica eliminar todas las relaciones que este tiene con otros recursos en el diagram de recursos, pero también implica eliminar todas las relaciones que este tiene con Tareas las cuales pertenecen a la vista de tareas (estas relaciones serán vistas más adelante

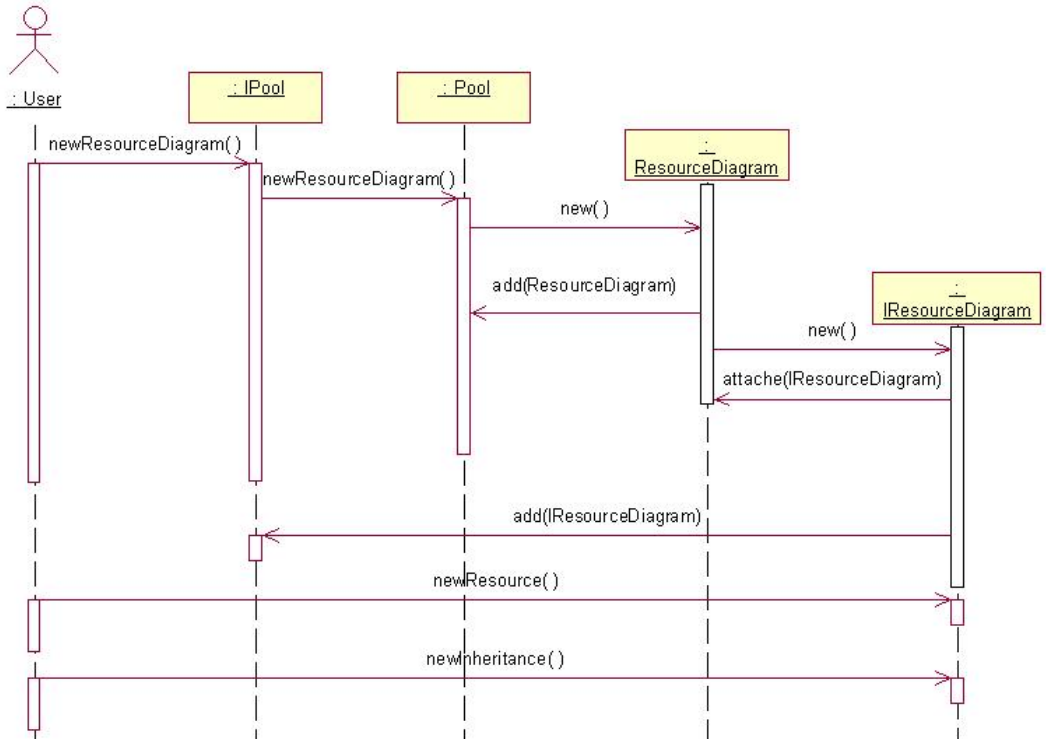


Figura 4.7: Diagrama de secuencia correspondiente a Crear un Diagrama de Recursos

cuando se explica la vista de Tareas). También es necesario eliminar el recurso de todos los diagramas de recursos en los cuales participa y eliminar sus interfaces gráficas. Cuando una instancia de *Resource* recibe el mensaje *delete*, debe:

- eliminar todas las relaciones con Tareas
- eliminar todas las relaciones con otros Recursos.
- eliminar la referencia de este recurso en los diagramas de recursos en los que participa
- eliminar sus interfaces gráficas.
- eliminar sus instancias.

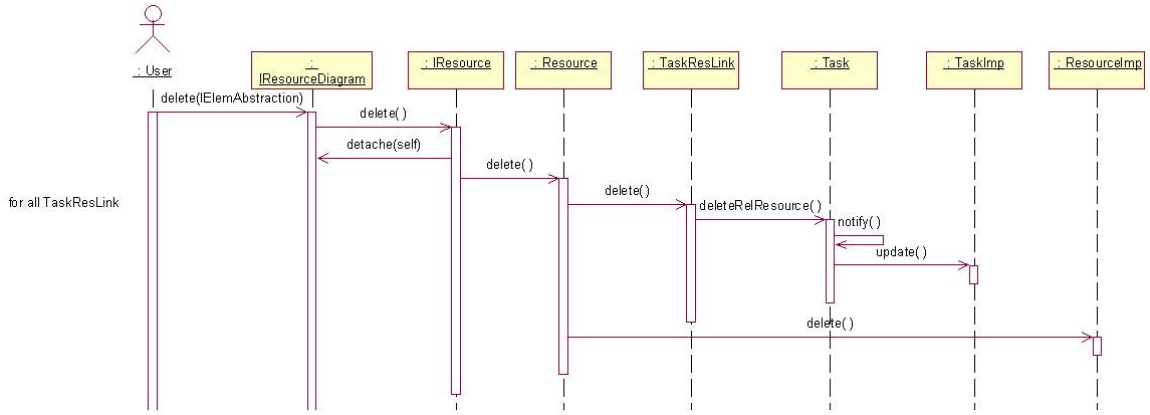


Figura 4.8: Diagrama de secuencia correspondiente a Eliminar un Recurso

La primera acción provocará una modificación de la definición de la tarea asociada al recurso con lo cual se dispara un mensaje *update()* a todas las implementaciones de ésta que, a su término eliminarán la relación que dejó de existir. La figura 4.8 muestra el diagrama de secuencia correspondiente al mensaje *delete* enviado a *Resource*. Por razones de simplicidad no se muestran las acciones llevadas a cabo por las *TaskImp*, dado que todavía no se ha hablado sobre las mismas. Más adelante se volverá sobre este tema.

4.1.2. Vista de Tareas

Esta vista representa la vista funcional de la organización. La vista de Tareas se enfoca principalmente en los procesos que se llevan a cabo en una empresa. Los procesos se definen como un conjunto de tareas relacionadas entre sí, directamente a través de relaciones temporales que definen el orden en que las mismas se ejecutan e indirectamente a través de los recursos que comparten. En esta vista, las Tareas podrán ser descompuestas en tareas más simples, representando de esta manera su descomposición funcional. El principal diagrama de esta vista es el *Modelo de tareas*, donde las tareas aparecen relacionadas entre sí y con recursos, reflejando las actividades llevadas a cabo

en una organización.

En el desarrollo de los modelos de tareas, se utilizan los conceptos de Clase e Instancia. Por lo tanto, se tendrá clases de tarea e instancias de tareas; clases de recursos e instancias de recursos; clases de relaciones tareas-recursos e instancias de estas relaciones.

Como se indicó anteriormente, una instancia es la representación de una ocurrencia o uso particular que se hace de un elemento genérico llamado clase. La clase representa un concepto que engloba las características y comportamientos comunes a un conjunto de objetos o instancias. Por ejemplo, si se considera la clase *camión*, este tiene ciertas características y comportamiento que son comunes a todas las instancias de camiones. El color, la capacidad, la velocidad, el número de patente, son características que todos los camiones tienen. Todos los camiones pueden trasladarse de un lugar de origen a uno de destino, pueden ser cargados o descargados, etc. Las características y el comportamiento asociado a la clase, tienen relación con el dominio que se está modelando. Por ejemplo, para un dominio de empresa como el que se está tratando, una característica importante de la clase camión puede ser la capacidad, para determinar por ejemplo qué productos puede transportar y en qué cantidades. Sin embargo la característica de la capacidad del tanque de nafta o el hecho de saber cuanta gasolina tiene un camión en un determinado momento puede ser irrelevante en este contexto. En una organización particular pueden existir varios camiones, por ejemplo una *instancia de camión* es el camión rojo de chapa patente ABC 123 cuya capacidad es 15 tn. Otra instancia posible es el camión verde de chapa patente ADD 333 cuya capacidad es de 40tn. Como se observa, una instancia asigna valores a las características definidas en la clase, en este caso, *color*, *número de patente* y *capacidad*.

Estos conceptos de clase-instancia utilizados en la Vista de Tareas, permiten realizar definiciones genéricas que conforman el vocabulario del dominio y definiciones

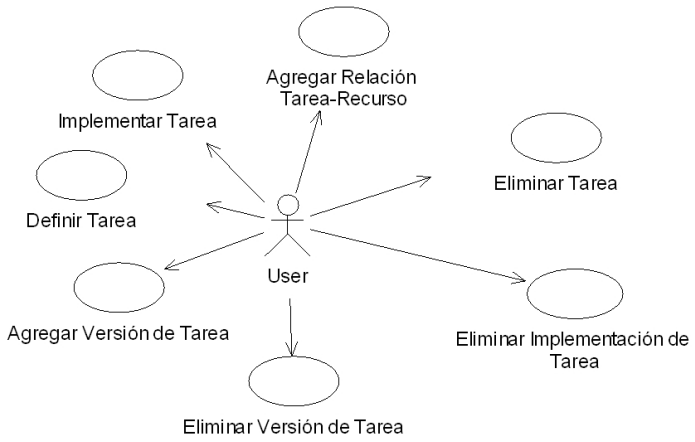


Figura 4.9: Diagrama de Casos de Uso de la Vista Tarea

particulares que definen en especial a una empresa. Las clases creadas formarán parte de la componente *Abstraction* mientras que las instancias de éstas serán parte del componente *Implementation*.

Para representar una descomposición, se utiliza el término *Versión de tarea*, ésta representa una forma particular de ejecutar una tarea. Así una tarea puede tener varias versiones asociadas cada una representando una forma particular de realizarla. Una versión puede variar de otra en el orden de ejecución de las tareas componentes, en los diferentes recursos que utilizan, en las condiciones de ejecución establecidas, etc. Las funcionalidades provistas en esta vista están representadas por los casos de usos que aparecen en la figura 4.9. Así es posible definir una tarea, crear una relación entre la tarea y un recurso previamente definido, agregar una versión de tarea, crear una instancia de una tarea, modificar una versión de tarea agregando o eliminando componentes a la misma.

El diagrama de clases de la figura 4.10 muestra los elementos del lenguaje que participan de esta vista.

La clase *Task* representa una tarea como está definida por el lenguaje Coordinates.

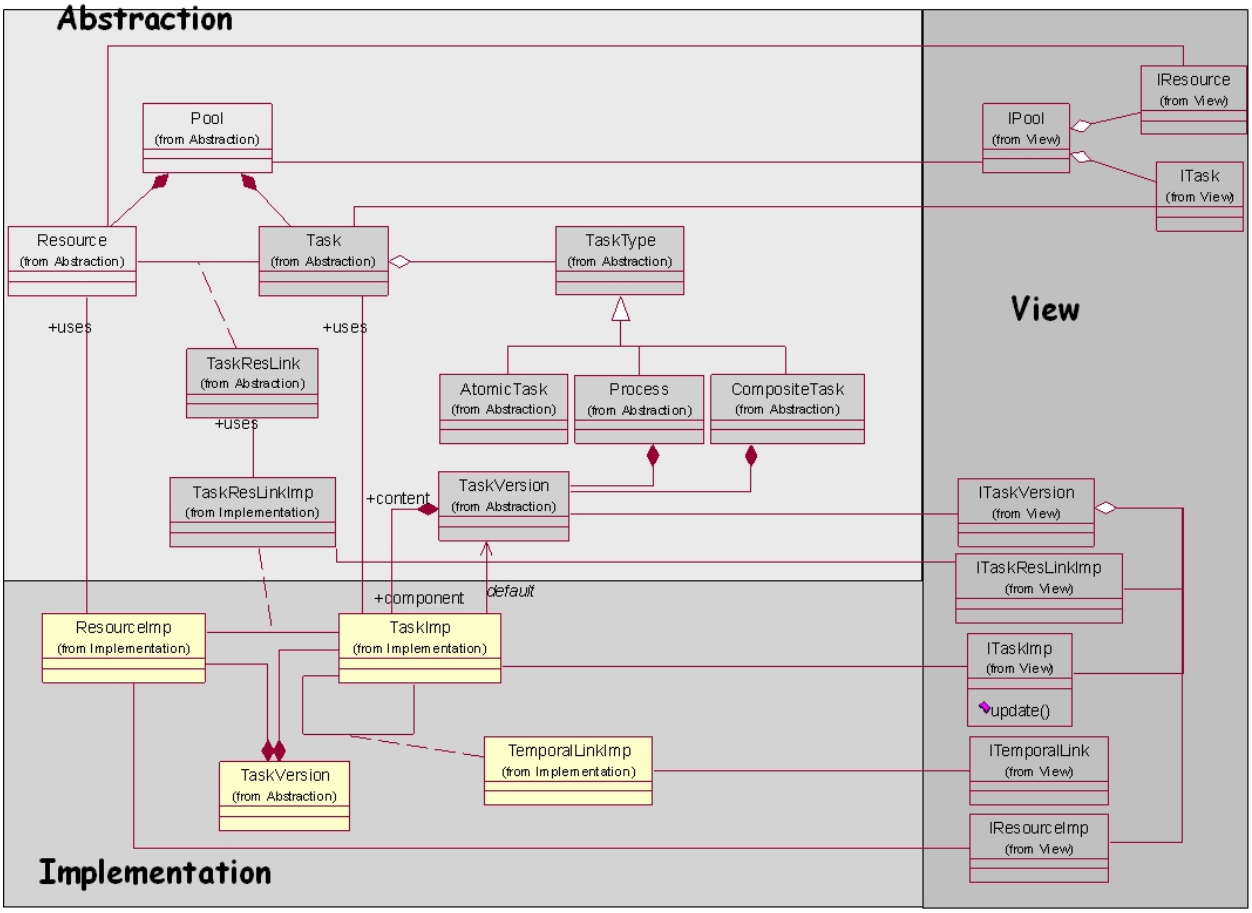


Figura 4.10: Diagrama de clases de la Vista de Tareas

Para representar los diferentes tipos de tareas que define el lenguaje seleccionado, se utilizó la clase TaskType, de esta forma una tarea tiene asociado un estado que identifica su tipo. Este diseño permite mantener en la clase *Task* las características generales de una tarea mientras el comportamiento asociado a los diferentes tipos se encapsula en las subclases concretas de la clase *TaskType*. Luego, *Task* delega en su tipo las acciones que varían de acuerdo a éste.

La clase *Task* tiene asociada una *TaskType* que representa su tipo. Las subclases de *TaskType* son: *AtomicTask*, representando una tarea simple, *CompositeTask*, representando una tarea compuesta y *Process* representando una tarea Proceso. Una tarea

simple o atómica (AtomicTask), es concebida como una tarea que no tiene una descomposición en subtareas. Una tarea compuesta (CompositeTask) por el contrario puede tener una o más descomposiciones representadas por sus versiones. Cada versión identifica una forma particular de llevar a cabo la tarea con respecto al conjunto de recursos que utiliza, al orden en que las subtareas son ejecutadas en la versión, etc. Una tarea compuesta se diferencia de un proceso (Process) en que la primera puede formar parte de la descomposición de otra tarea de nivel superior, mientras que el proceso solo puede ser la raíz de una estructura de descomposición. La ventaja de usar este diseño para representar tareas es la posibilidad de alterar en tiempo de ejecución el tipo de una tarea. La necesidad de esta característica está basada en el proceso incremental de desarrollo del modelo de empresa soportado por esta arquitectura. Cuando se está en proceso de desarrollo del EM, es posible que una tarea se conciba inicialmente como una tarea simple, pero en procesos subsiguientes, la misma puede ser vista como una tarea compuesta descripta a través de otras tareas mas simples. Esta forma dinámica que la tarea tiene de cambiar su tipo, le permite mantener su definición conceptual y el conjunto de servicios que la misma brinda (relaciones con recursos, relaciones con otras tareas, atributos, etc.) mientras cambia su estructura (descomposición jerárquica) y su comportamiento.

El diagrama de estado de la figura 4.11 muestra los estados posibles de una tarea y las transiciones producidas por las acciones llevadas a cabo sobre la tarea. Se consideraron como estados los distintos tipos de tareas que están definidas en Coordinates. Como se observa, una tarea puede estar en tres posibles estados: *AtomicTask*, *Process* y *CompositeTask*. Inicialmente, cuando se define la misma, el estado asociado es *AtomicTask*. Luego, si se le agrega una versión de tarea, para descomponerla en subtareas puede suceder que: (i) la tarea no tenga implementaciones, entonces pasa al estado *process*, es decir, no participa en la descomposición de otras tareas, o (ii) la tarea

tiene implementaciones, entonces el estado pasa a ser *CompositeTask*. Una tarea pasa del estado *CompositeTask* al estado *Process*, si se eliminan todas sus implementaciones en cuyo caso, la tarea no participa en la descomposición de otras tareas y representa una raíz en esta estructura de descomposición. Por otro lado, se pasa del estado *Process* al estado *CompositeTask* cuando se agregan implementaciones a la definición de la tarea. En este caso, la tarea deja de ser una raíz de descomposición para pasar a ser parte de una descripción de una tarea de nivel superior. Finalmente, si la tarea se encuentra en el estado *CompositeTask*, y todas sus versiones son eliminadas pasa al estado *AtomicTask*. Este ciclo de vida rige las diferentes asociaciones que una instancia de *Task* tendrá durante su vida.

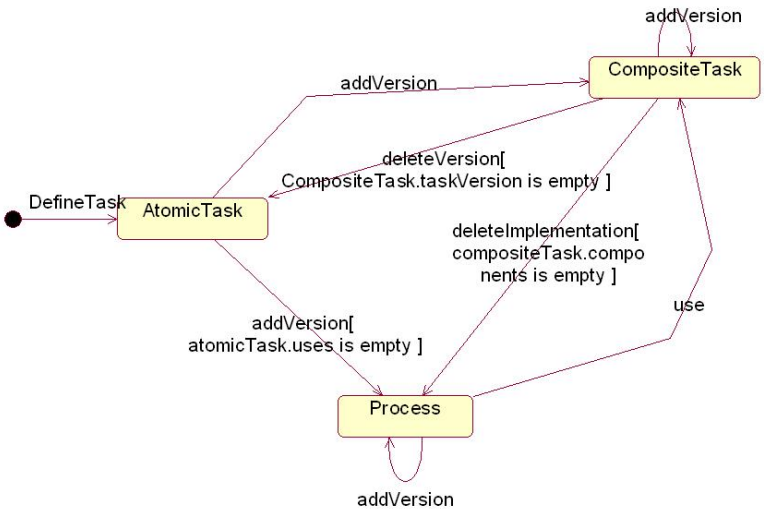


Figura 4.11: Ciclo de vida de Tarea

TaskVersion (figura 4.10) corresponde a la versión de tarea y representa una descomposición particular de la misma. Esta clase se encuentra asociada a *CompositeTask* y a *Process* por ser éstos los posibles tipos de tareas que soportan descomposiciones. La *TaskVersion* es especificada en términos de diferentes elementos de implementación, cada uno capturando un comportamiento específico de algún concepto definido en *Abs-*

traction, el cual asume como consecuencia de participar en esta descomposición. Más específicamente una *TaskVersion* queda representada como una colección de implementaciones de tareas, recursos, relaciones tarea-recursos y relaciones temporales entre las tareas participantes. Estas implementaciones que conforman la versión de tarea, son parte de la componente *Implementation*. Así, una versión de tarea significa la forma de llevar a cabo una tarea particular en función de otras tareas, instanciadas con el fin de representar una forma particular de la misma en ese contexto.

La clase *TaskResLink* representa la relación entre *Task* y *Resource* e implementa el concepto de relación tarea-recurso definida en Coordinates. Estas relaciones identifican la forma en que una tarea usa a un recurso según la semántica definida en Coordinates. *TaskResLink* se especializa en *use, create, modify, delete, consume* y *produce*. Así, una relación *usa* (use) establece que el recurso participa como una herramienta en la ejecución de la tarea, durante la ejecución el recurso puede cambiar de estado, pero al finalizar la misma deja al recurso en el estado en que lo encontró. La relación *crea* (create) establece que el recurso no existía antes de la ejecución de la tarea y comienza a existir cuando esta finaliza. La relación *elimina* (delete) es la inversa de crea. Las relaciones *consume/produce* (consume/produce) indican que la cantidad del recurso cambia disminuyendo/incrementando en una cantidad definida en la relación. Principalmente esta relación se utiliza para recursos que son contables. La relación *modifica* indica un cambio de estado en el recurso que participa.

La clase *TaskImp* representa las implementaciones de las tareas. Cuando una tarea es implementada, implica que la misma es usada en un contexto particular para lo cual adquiere ciertas características individuales que la diferencian de otras implementaciones. Las implementaciones participan de las versiones de tareas. Así cada vez que una tarea es seleccionada del repositorio para participar en una versión, una implementación de la misma es creada. Debido a que las relaciones de la tarea con los recursos

que la misma utiliza en su ejecución forman parte de la definición de ésta, cuando se selecciona una tarea para implementar, esta acción implica la implementación de sus relaciones. El vínculo *default* que relaciona una *TaskImp* con una *TaskVersion* identifica la forma que adopta la implementación en su contexto. En otras palabras, si una *Task* tiene varias formas de ejecutarse representadas por sus *TaskVersion*, luego cuando una *TaskImp* es creada para participar en un contexto dado, ésta debe asumir una de todas las formas que tiene *Task*, el vínculo *default* es el que determina cual de ellas adopta la implementación específica.

La clase *TemporalLink* representa las relaciones temporales que pueden vincular dos tareas directamente, indicando el orden de ejecución de las mismas. Estas relaciones tiene correspondencia con las relaciones definidas por Allen (Allen, 1984): antes que, solapa, encuentra, finaliza, comienza y durante (before, meets, overlaps, equals, finishes, start, during).

A continuación se describirá cómo definir un EM a través de esta vista desarrollando los casos de usos mostrados en la figura 4.9.

Caso de uso: Definir tarea. Definir una tarea significa crear un nuevo concepto, el cual es un componente de modelado que formará parte en el futuro del modelo de empresa que se desarrolle. Como tal, esta definición permanecerá en el repositorio, *Pool*, para ser seleccionada como participante de algún modelo de tareas. La definición de la tarea no solo implica la asignación de un nombre y la especificación de sus características particulares como puede ser el tiempo de ejecución, sino también es necesario indicar el conjunto de recursos con los cuales está relacionada, es decir aquellos que participan de la tarea. Es posible crear una tarea desde la interfaz gráfica del repositorio *IPool*. El estado por defecto que asume la tarea creada es *AtomicTask*, la misma irá evolucionando a otros tipos con las sucesivas interacciones, según fue explicado anteriormente. La

Figura 4.12 muestra el diagrama de secuencias para crear la tarea desde esta interfaz gráfica. Cuando se selecciona la acción de crear una nueva tarea, se crea una instancia de *Task*, la cual tendrá asociado como estado una instancia de *AtomicTask*. Seguidamente, la tarea crea su interfaz gráfica *ITask* quien se agrega como observador con el método *attach(self)*, y con la acción *addComponent(self)* se agrega como contenido del repositorio *Pool*. Finalmente, se seleccionan los recursos que participarán de la tarea. Para ello se debe seleccionar un recurso y un tipo de relación que lo vincula con la tarea creada previamente, una instancia de *TaskResLink* se crea, que vincula el recurso seleccionado con la tarea en cuestión.

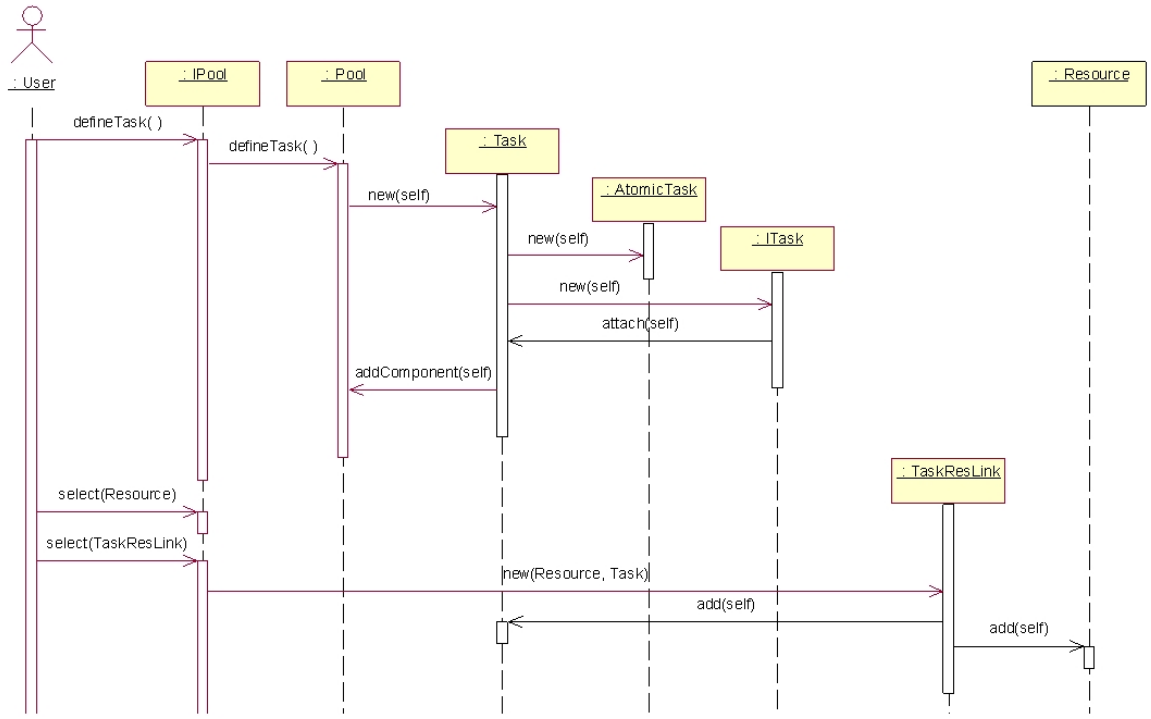


Figura 4.12: Diagrama de secuencia correspondiente a Definir una Tarea

A medida que se avanza en el diseño, nuevas relaciones entre tareas y recursos pueden generarse. De esta manera, una vez que la tarea se crea, su definición podrá ir cambiando agregando o eliminando nuevas relaciones. Se verá más adelante cómo el agregado o

eliminación de relaciones entre una tarea y un recurso afecta las implementaciones de esta tarea. El mecanismo de actualización (observer) que existe entre una tarea y sus implementaciones se pondrá en funcionamiento cada vez que modificaciones en la definición de la tarea sucedan.

Caso de Uso: Agregar una Versión de Tarea. Una Versión de tarea, como se indicó, representa una forma particular de llevar a cabo una tarea. Una tarea puede tener asociada varias versiones de tareas diferentes, cada una indicando una manera de ejecutarla. La versión de tarea puede verse también como una descomposición de una tarea en subtareas. Así, una versión de tarea está formada por un conjunto de tareas más simples y representan una descomposición funcional de la tarea asociada a la versión. Las versiones de tareas se describen a través de *implementaciones*, es decir *instancias* de ciertos elementos definidos en el dominio de abstracción y que forman parte del vocabulario de modelado. Cada Versión de Tarea encapsula una combinación particular de implementaciones de tareas, recursos, relaciones temporales y relaciones tarea-recurso relevantes, todas ellas en un contexto dado. Cada vez que una tarea **T**, definida en el dominio de abstracción, participa en la descripción de un tarea más compleja **P**, una instancia de **T**, **IT**, se crea para formar parte de la versión de tarea que representa la descomposición de **P**. Esta instancia **IT** tendrá características particulares como ser entre otras cosas, su tiempo de ejecución en el contexto de la tarea **P**, el conjunto de recursos que participan y las relaciones temporales con las otras instancias que forman parte de la misma versión de **P**.

Cuando una versión de tarea se agrega a una tarea atómica, provoca que la tarea cambie de tipo, pasando a ser una tarea compuesta, si la misma pertenece a otras implementaciones, o una tarea proceso en caso contrario. El diagrama de secuencia de la figura 4.13 muestra la creación de una versión de tarea asociada a una tarea sim-

ple, es decir una tarea cuyo tipo asociado es *AtomicTask*. En este diagrama, la tarea evoluciona al tipo *CompositeTask*. La necesidad de descomponer una tarea puede ser requerida tanto desde la interfaz gráfica de la tarea *ITask* que aparece en *IPool*, como de la interfaz gráfica de algunas de sus implementaciones *ITaskImp* que pertenece a algún diagrama. En el ejemplo presentado, una tarea *Task* (cuyo tipo asociado es *AtomicTask*) es seleccionada para enviarle el mensaje *AddVersion()*. La tarea delega en su tipo la acción recibida. Como consecuencia, se produce un cambio de tipo, el cual se ve reflejado en el conjunto de mensajes producidos entre *AtomicTask* y *CompositeTask*. Así, cuando *AtomicTask* recibe el mensaje *addVersion*, crea una instancia de *CompositeTask*, el cual pasa a ser el nuevo tipo de la tarea. A través del mensaje *setState()* el tipo *CompositeTask* solicita a *Task* ser reconocido como el nuevo tipo de la tarea. Seguidamente, *AtomicTask* delega en el nuevo tipo la acción de agregar la versión. Luego, *CompositeTask* crea la versión de tareas enviando el mensaje *new()* a *TaskVersion*. Como consecuencia del patron *Observer*, *Task* notifica del cambio de estado a sus interfaces gráficas e implementaciones enviando el mensaje *update()*.

Caso de uso: Implementar una Tarea. Implementar una tarea significa seleccionarla para ser usada en un contexto específico, más concretamente en una Versión de tarea que se está diseñando. Como tal, el uso de la tarea implica la concretización de sus características, a saber: recursos que utiliza, versión asociada en caso de ser una tarea compuesta, valores de atributos, como ser su tiempo de ejecución, valores de las relaciones con recursos, como por ejemplo en qué cantidad una tarea consume a un recurso, etc. El diagrama de secuencia de la figura 4.14 muestra la implementación de una tarea. En este ejemplo, la interfaz gráfica de la tarea *ITask*, recibe el mensaje *use()*, dicho requerimiento es delegado en *Task*, la cual crea su implementación y la asocia a la versión de tareas correspondiente. Al crearse la implementación de la tarea,

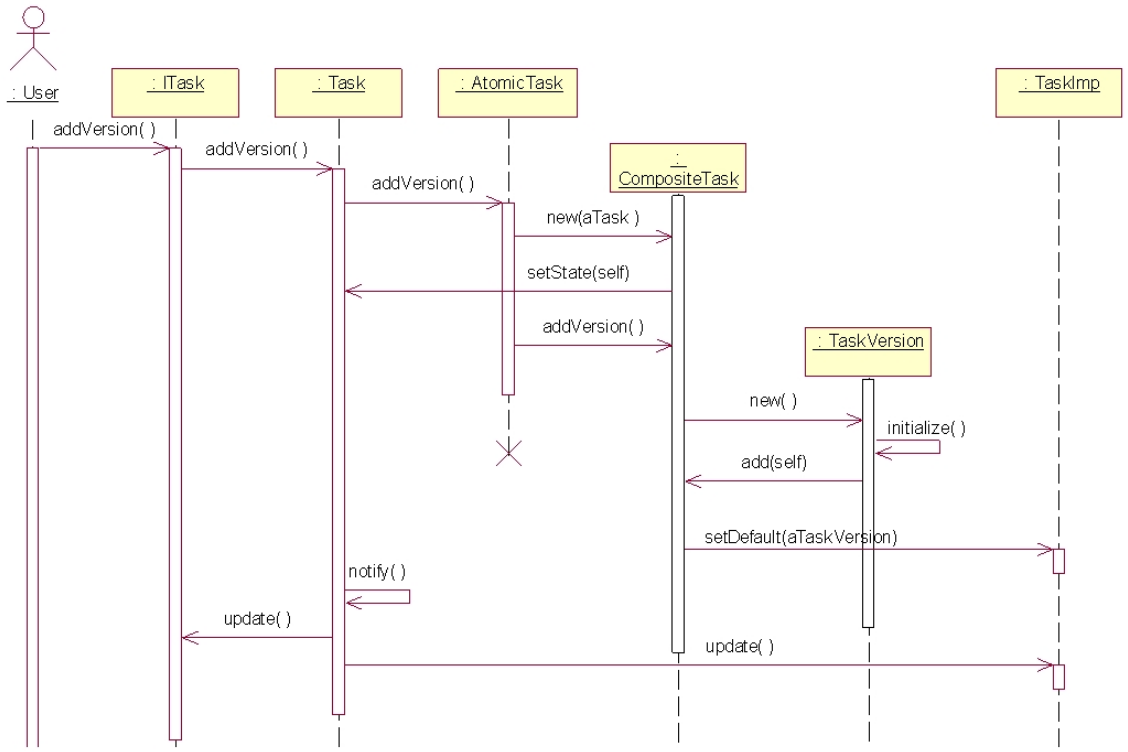


Figura 4.13: Diagrama de secuencia correspondiente a Agregar Versión de Tarea

las implementaciones de sus relaciones con recursos también son creadas y agregadas a la versión. Cuando la clase *TaskResLink* recibe el mensaje *use()* para crear su implementación, esta relación obtiene las implementaciones existentes en la versión a la cual pertenece, así, crea sus implementaciones usando las implementaciones de los recursos existentes. En caso que no existan en esa versión implementaciones de los recursos, solicitará al recurso que cree sus implementaciones correspondientes. Es de destacar que al ser el proceso de diseño iterativo e incremental caben muchas posibilidades de cómo definir una implementación. Cuando se crea una implementación de tarea, ésta estará relacionada a instancias de recursos existentes, pero a medida que el proceso de diseño del modelo avanza, estas relaciones pueden cambiar. Nuevas implementaciones de recursos pueden aparecer en la versión con los cuales asociar las tareas participantes.

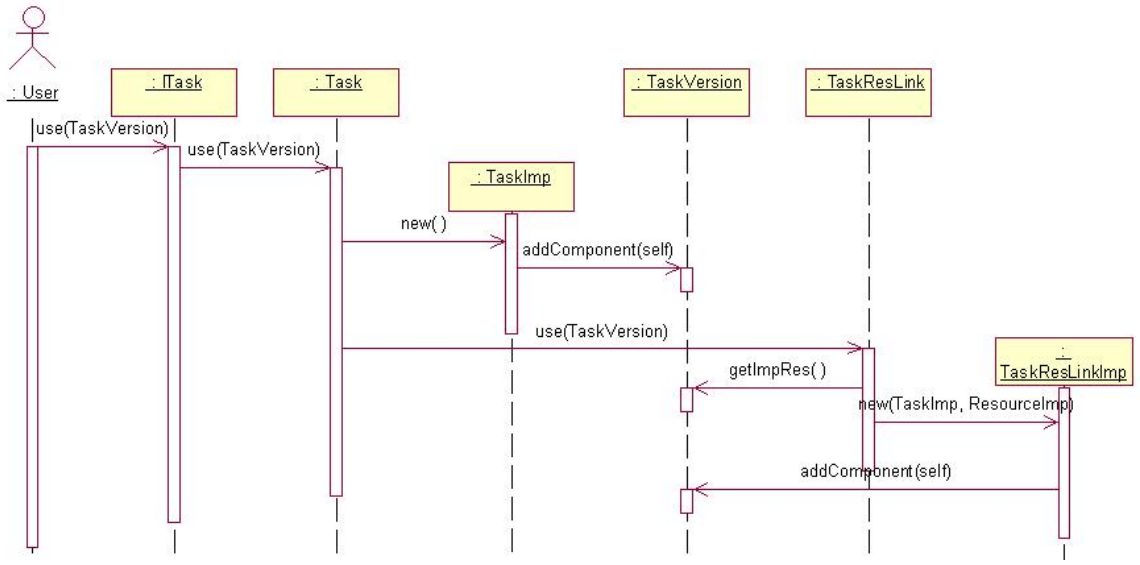


Figura 4.14: Diagrama de secuencia correspondiente a Implementar una tarea

Caso de uso: Agregar relación Tarea-recurso. Este caso de uso se presenta con el objetivo de mostrar el mecanismo de actualización. La creación de una nueva relación entre una tarea y un recurso implica un cambio en la definición de la tarea, el cual debe manifestarse tanto en sus interfaces gráficas como también en sus implementaciones. El diagrama de secuencia de la figura 4.15 muestra las interacciones llevadas a cabo.

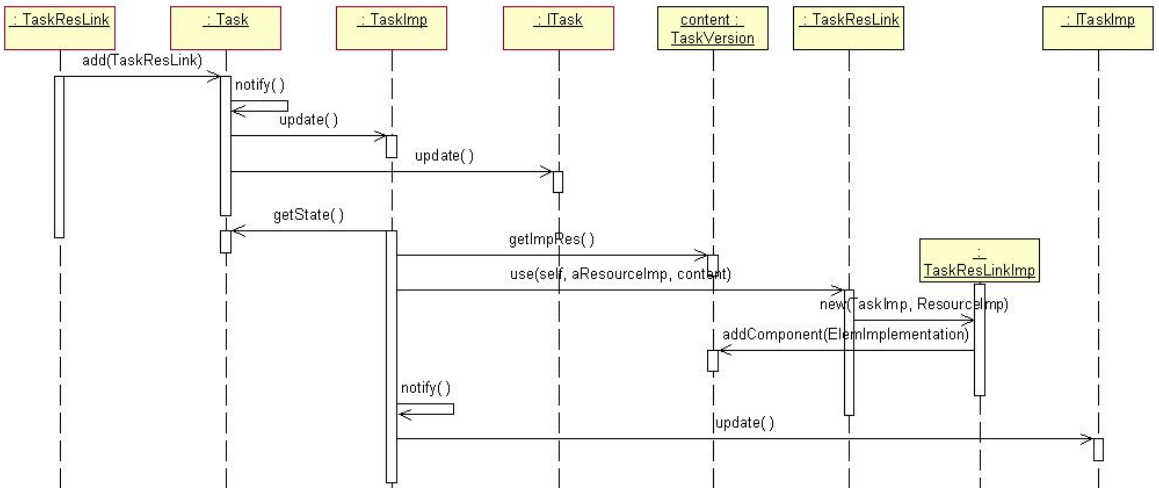


Figura 4.15: Diagrama de secuencia correspondiente a agregar una relación tarea-recurso

Cuando la tarea recibe el mensaje de agregar una nueva relación *TaskResLink*, notifica a todos sus observadores enviando el mensaje *update()*. En el diagrama solo se muestra cómo actúa una implementación de la tarea que se modificó al recibir este mensaje. Primeramente, la implementación *TaskImp*, solicita a *Task* el estado de la misma enviando el mensaje *getState()*. Para poder actualizarse, debe seleccionar una implementación del recurso relacionado que exista en la versión de tarea de la cual la *TaskImp* forma parte, referenciada como *content* en la figura. Luego la *TaskImp* envía el mensaje *use()* a la *TaskResLink* agregada para que ésta cree su implementación, seguidamente se crea una instancia de una *TaskResLinkImp* que representa la nueva relación agregada en *Task*. Para finalizar la actualización, la *TaskImp* debe enviar los mensajes de actualización a sus observadores para que estos actúen en consecuencia.

4.1.3. Vista Dinámica

La Vista dinámica está centrada en el desarrollo de los diagramas de transición de estados (DTE) asociados a los recursos. Estos diagramas son usados para representar las transformaciones que sufren los recursos al participar en la ejecución de las tareas. Si se tiene en cuenta que un recurso puede participar en varias tareas, cada una de ellas lo transformará de una manera distinta, esto quiere decir que un DTE abarca todas las transformaciones posibles del recurso, representando en cierta forma el comportamiento del mismo en todo su ciclo de vida. Para crear un DTE, es necesario indicar los estados, las transiciones de estados y las acciones o eventos que causan estos cambios, así como también restricciones que deben cumplirse para que la transición tenga efecto cuando un evento ocurre. Un *estado* puede ser vistos como representando situaciones abstractas en el ciclo de vida de un recurso, o como una invariante temporaria del mismo. Por ejemplo, un recurso *impresora* puede estar en el estado *libre* permaneciendo en ese estado por un

tiempo indefinido hasta que un evento *comenzar a imprimir* ocurra. Las *transiciones* de estados son los caminos que determina hacia qué estado se puede evolucionar desde el estado actual, dado que ocurrió un evento. Éstas representan transformaciones o cambios discretos como consecuencia de la ocurrencia de acciones o eventos. Por lo tanto, las transiciones tendrán asociadas acciones que se llevan a cabo sobre el recurso para modificarlo de alguna manera y permitir así el cambio de estado. Estas acciones son consideradas eventos, en el sentido que no se les asigna un tiempo sino que son instantáneas. Siguiendo con el ejemplo del recurso *impresora*, la aparición del evento *comenzar a imprimir* causa que la transición entre el estado *libre* y *ocupado* tenga efecto. Para el caso particular de esta vista, las acciones que modifican los recursos son las tareas, pero estas no pueden ser consideradas como eventos dado que tienen una duración asociada. Para poder salvar este inconveniente se definieron los eventos (*startTrigger*) y (*endTrigger*) correspondientes a las acciones de iniciar y terminar una tarea. De esta forma, si el recurso *impresora* se encuentra en el estado *libre*, y la tarea *imprimir* comienza a ejecutarse, se generará un evento correspondiente al inicio de esta tarea que provocará un cambio de estado en el recurso. Al finalizar la tarea *imprimir*, otro evento será generado para provocar un cambio en el recurso que participa de la tarea que finalizó. Durante la ejecución de la tarea, el recurso estará en un estado intermedio (se podría resumir diciendo que el recurso está en el estado *participando de la tarea*).

La vista dinámica permite tener una perspectiva distinta en las dependencias entre las tareas, ahora centrada en su relación con los recursos. Cuando varias tareas interactúan compartiendo recursos, la dependencia entre una y otra puede observarse a través de los estados en que dejan a los recursos. Para que una tarea pueda ejecutarse, necesita como condición que los recursos que usa estén en un determinado estado, por ejemplo para poder comenzar a *imprimir* es necesario tener la *impresora* en estado *libre*.

En este sentido, se puede decir que el estado *libre* es una *precondición* para la tarea *imprimir*. De igual manera, cuando una tarea termina de ejecutar, causa una transición de estado en el recurso, este nuevo estado se lo conoce como *post-condición* de la tarea. Así, la post-condición de una tarea puede ser la precondición de otra, dando como resultado un encadenamiento entre las tareas dirigido ya no por relaciones temporales sino por la dependencia que se genera al utilizar los mismos recursos.

La figura 4.16 muestra los casos de uso definidos para esta vista. Así, es posible crear un diagrama de transición de estado asociado a un recurso existente, crear o eliminar estados de recursos y crear o eliminar transiciones entre estos estados.

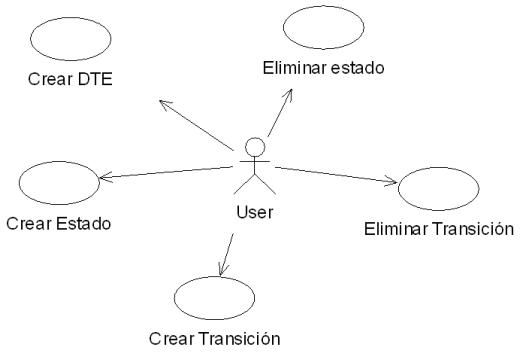


Figura 4.16: Diagrama de casos de uso correspondiente a la Vista Dinámica

Para dar soporte a las funcionalidades definidas anteriormente, el diagrama de clases de la figura 4.17 muestra los elementos del lenguaje necesarios en esta vista.

Para representar el ciclo de vida de los recursos, la vista dinámica utiliza el formalismo de statechart propuesto por Harel (Harel, 1987); (Harel y Gery, 1996). Así, la evolución de un recurso es representada como un conjunto de estados y transiciones entre estos estados. Como se observa, los recursos tienen asociado su ciclo de vida representado por la clase *ResLifeCycle*, el cual contiene un conjunto de estados del recurso llamados *ResState*. Los estados están conectados unos con otros a través de

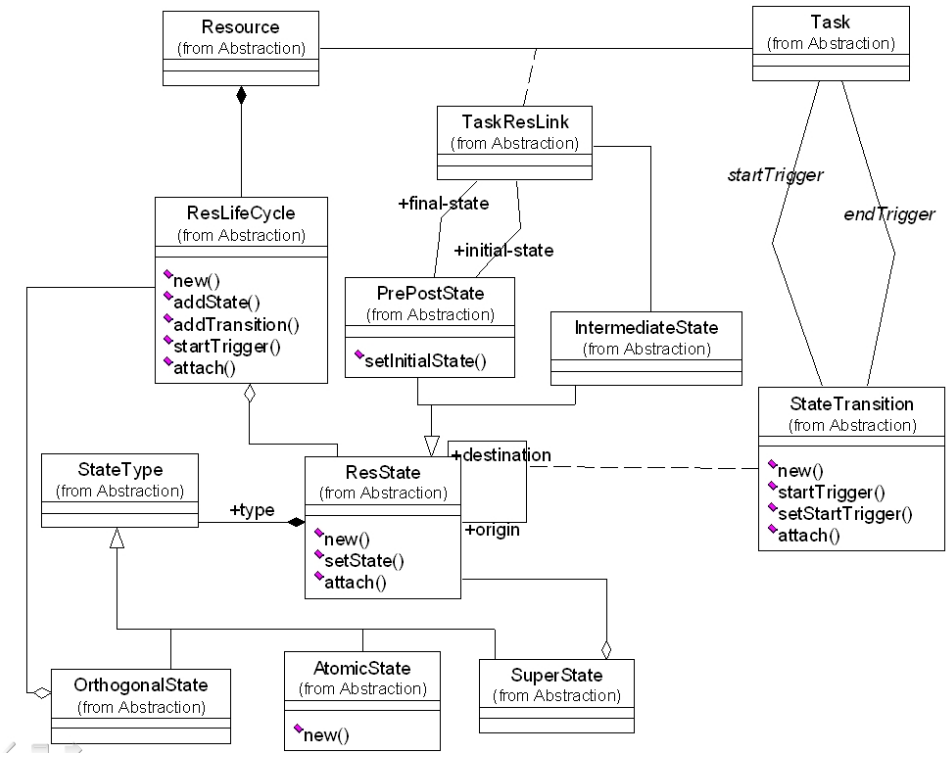


Figura 4.17: Diagrama de clases correspondiente a la vista Dinámica

transiciones, *StateTransition* las cuales asocian un estado origen con un estado destino indicando que existe un camino dirigido entre estos estados. Estas transiciones tienen relación con las tareas que causan dicha transición. Como se explicó anteriormente, los eventos que causan transiciones de estados en los recursos son los inicios y fin de la ejecución de las tareas. Por lo tanto, la clase *StateTransition* está asociada a la clase *Task* a través de dos vínculos: *startTrigger* y *endTrigger*, indicando el comienzo y fin de la tarea que provoca dicha transición. Cada vez que una tarea está asociada a un recurso a través de una relación *TaskResLink*, esa asociación identifica: (i) un estado **inicial** del recurso, indicando el estado en que el recurso debe encontrarse para que la tarea pueda ser ejecutada, lo cual se establece mediante la relación *inicial-state* entre *TaskResLink* y *PrePostState*; (ii) un estado **intermedio**: englobando un conjunto de estados por los que el recurso transita mientras la tarea se ejecuta, este estado se expresa mediante la

clase *IntermediateState*; (iii) un estado **Final**: representando el estado en que queda el recurso al finalizar la tarea, este tipo de estado está representado por la asociación *final-state* entre *TaskResLink* y *PrePostState*.

El modelo utiliza tres tipos de estados de acuerdo con el formalismo de los state-chart: estados ortogonales (*OrthogonalState*), estados atómicos (*AtomicState*) y super estados (*SuperState*). Un estado atómico es aquel que no permite una descomposición en subestados, mientras que el super estado y el estado ortogonal sí lo permiten. Estos dos últimos representan estados con semánticas diferentes: el primero de ellos, super estado, encapsula la semántica de una disyunción *Xor*, esto es, en un determinado instante, un recurso puede estar en uno solo de los sub-estados que conforman el super estado. Por el contrario un estado ortogonal, tiene la semántica de una conjunción *and*, donde el recurso puede encontrarse en todos y cada uno de los estados que componen al estado ortogonal. Los componentes de un estado ortogonal se representan a su vez como diagramas de transición de estados. Este tipo de estado es utilizado para representar el ciclo de vida de recursos compuestos, de esta manera cada estado ortogonal representa el ciclo de vida de los componentes independientes. Un estado ortogonal también es útil para representar las diferentes perspectivas con que un recurso puede ser visto. Como consecuencia del proceso iterativo e incremental asociado al modelado de empresa, los estados asociados a un recurso pueden ir evolucionando a través de los diferentes tipos durante el proceso de modelado de la organización. Es por esta razón que se utilizó el patrón de diseño *State* para soportar dicha evolución. Se verá a continuación en más detalles los casos de uso planteados en esta vista.

Caso de uso: Crear DTE. Un diagrama de transición de estado se crea asociado a un recurso, indicando el ciclo de vida para todas sus instancias al participar de las diferentes tareas. Para crear este diagrama es posible seleccionar un recurso de la vista

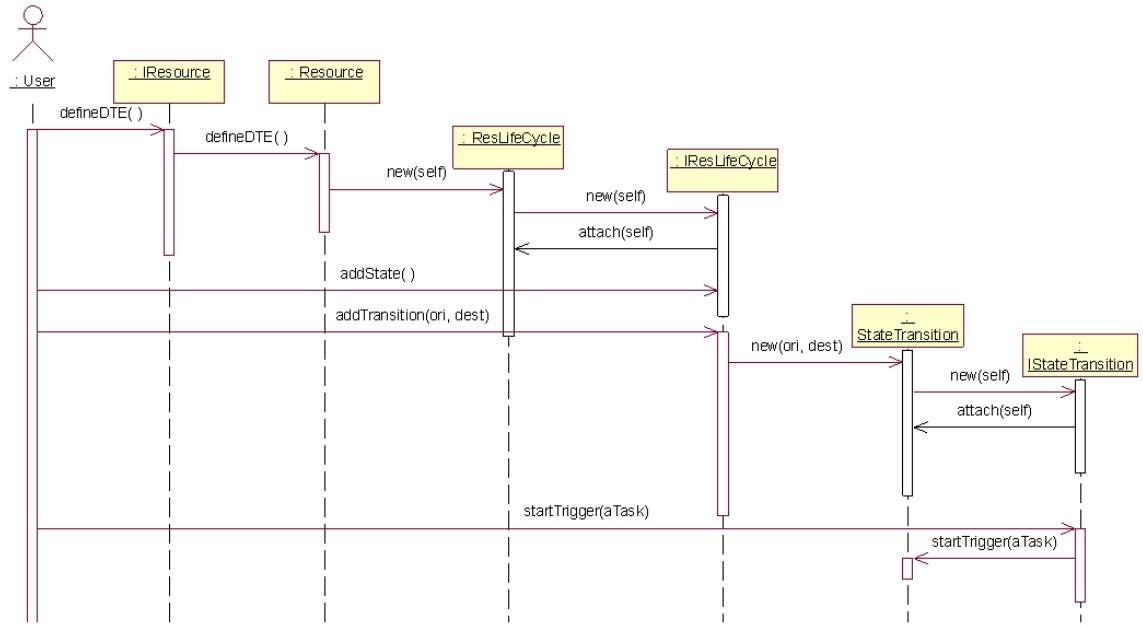


Figura 4.18: Diagrama de secuencia para el caso Crear un DTE

del dominio y asociarle un diagrama, pero también es posible seleccionar una instancia de un recurso, en cuyo caso, la instancia delega en su clase la acción de crear el diagrama de transición de estado. El gráfico de la figura 4.18 muestra el conjunto de interacciones llevadas a cabo cuando se selecciona un recurso desde la vista del dominio para definir su ciclo de vida.

Así, el recurso crea un diagrama de transición de estado, el cual a su término crea su interfaz gráfica. Inicialmente el diagrama se crea vacío, luego, con las sucesivas interacciones con el usuario, se irán agregando estados y transiciones al diagrama a medida que se avance en el proceso de modelado. En el diagrama de secuencia se muestra la creación de una transición entre dos estados identificados como origen y destino. Una vez creada la transición, es posible identificar una tarea como el evento que causa la transición. En el diagrama de secuencia se muestra el envío del mensaje *startTrigger(aTask)* a la transición, indicando que dicha transición tendrá efecto con el inicio de la tarea *aTask*.

Esta secuencia de pasos será ampliada más adelante cuando se explique el caso de uso *agregar transición*.

Caso de uso: Crear Estado. Cuando un estado se crea, forma parte de una diagrama de transición de estado correspondiente a un recurso (figura 4.19. El estado nuevo se asume que es atómico, luego podrá ir cambiando en caso que se agreguen subestados u otros diagramas de transición de estados como componentes.

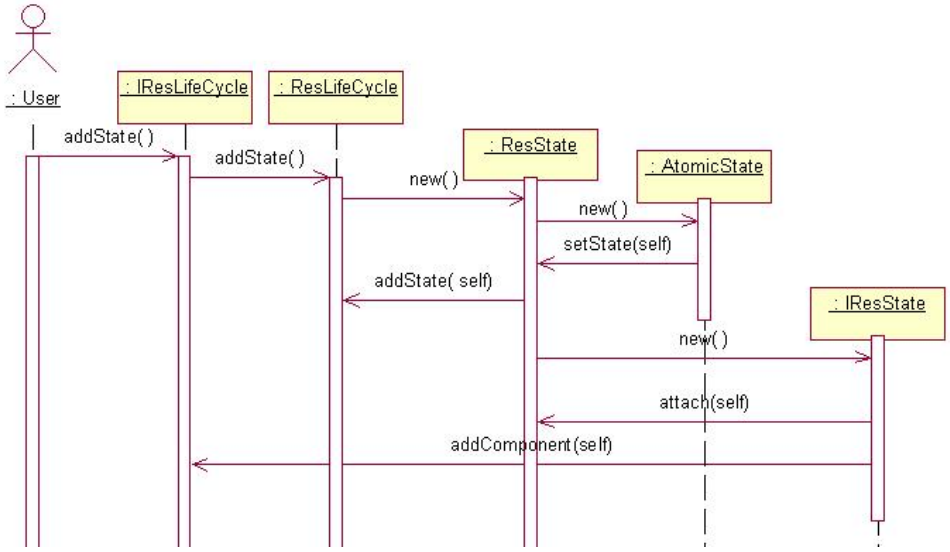


Figura 4.19: Diagrama de secuencia correspondiente a agregar un estado

El nuevo estado, tiene su interfaz gráfica asociada la cual forma parte del la interfaz gráfica del diagrama. A través del mensaje *attach(self)*, la instancia de *IResState* se registra como observador del estado que se acaba de agregar. De esta forma, cuando el estado cambia de tipo, inmediatamente la interfaz gráfica se modifica para mostrar dicho cambio.

Caso de uso: Agregar una Transición. Agregar una transición en un diagram de transición de estado implica seleccionar los estados origen y destino de dicha transición

como así también el evento que la causa. Como se explicó anteriormente, existen dos posibles eventos que pueden asociarse con las transiciones de estados: el comienzo y el fin de una tarea.

El diagrama de la figura 4.20 muestra las interacciones entre los distintos objetos intervinientes para crear una transición.

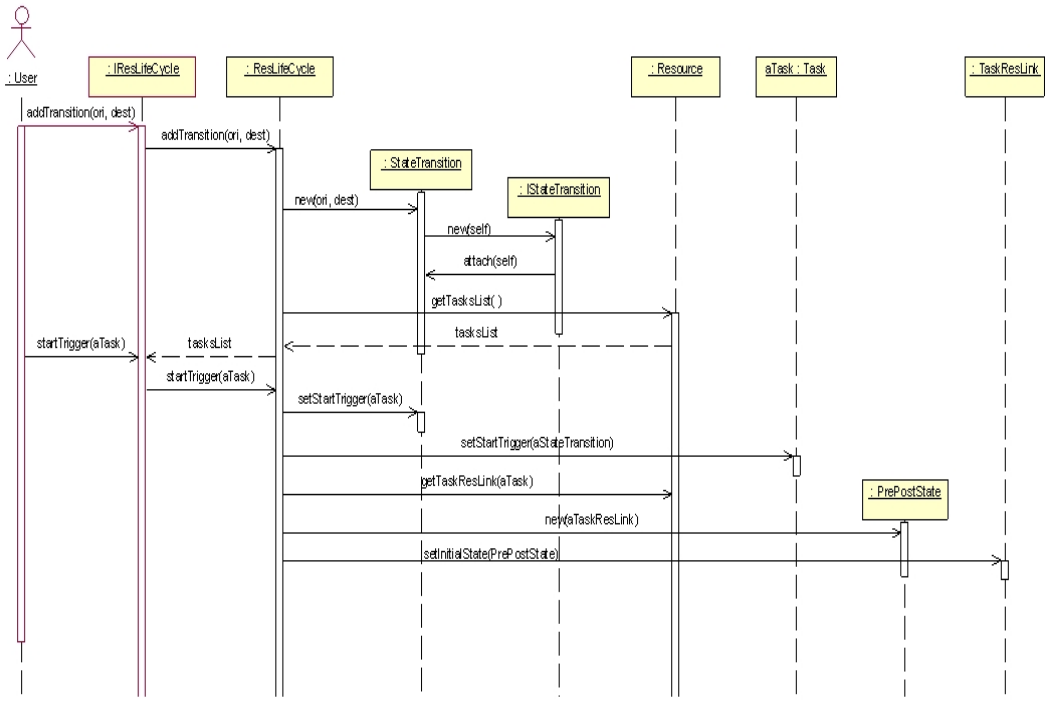


Figura 4.20: Diagrama de secuencia correspondiente a Crear una Transición

Se identifica el estado origen y destino de la transición que se crea. La misma es creada y su interfaz gráfica es agregada como parte del diagrama que se está generando. Luego es necesario indicar el evento que causa la transición. Para ello el usuario selecciona una tarea de todas aquellas que utilizan al recurso. Esta tarea es la que se asocia con la transición creada y se identifica si la misma corresponde a un evento *startTrigger* o *endTrigger*. En el gráfico mostrado, se tomó el ejemplo del evento *startTrigger*. De esta forma, cuando se identifica el evento es necesario establecer el vínculo entre la

relación tarea-recurso (TaskResLink) y el estado origen (PrePostState) de la transición para indicar que dicho estado es una precondición para la tarea seleccionada. De igual manera se procede en el caso de seleccionarse el evento *endTrigger*, en cuyo caso se asocia la relación tarea-recurso a través del vínculo *final-state*, indicando que el estado *PrePostState* es el estado en que la tarea deja al recurso una vez que la misma finaliza su ejecución.

Esta especificación sirvió de base para el desarrollo de una herramienta prototipo que da soporte a la construcción de los modelos de empresa de acuerdo a los lineamientos presentados. Dicho prototipo, llamado Coordinates Workbench, se encuentra descripto en el Anexo A.

4.2. Ejemplo

En esta sección se plantea un ejemplo de un proceso de suministro involucrando una organización de producción, la cual entrega harina industrial a su centro de distribución en el área geográfica Santa Fe. Una vez a la semana un empleado de logística toma las órdenes de suministro (OS) requeridas por el centro de distribución del área Santa Fe y prepara el plan de distribución para la siguiente semana. Un sistema de plan de distribución, genera el plan de distribución (PD) el cual es enviado vía e-mail al área de despacho. Una nueva orden de suministro es aquella OS que ha sido recibida durante la semana, mientras que una OS demorada es aquella OS que no ha podido ser satisfecha con el PD previo y por lo tanto tiene mayor prioridad. Esto quiere decir que las OS demoradas son procesadas primero. Todas las mañanas un empleado de despacho imprime una orden de entrega (OE, 4 copias) para cada centro de distribución de acuerdo al PD de la semana. El empleado de reparto, toma la copia 1, 2 y 3 de la OE junto con la mercadería, "*harina industrial*". En esta organización, existe un camión

que está destinado a las entregas de mercadería al centro de distribución Santa Fe, para su referencia se dirá que es el "*camión Santa Fe*". La copia 4 de la OE es temporalmente almacenada en la zona de despacho. Mientras la mercadería es descargada del camión en el centro de distribución correspondiente, la misma se chequea por problemas de calidad y cantidad, y las copias 1 y 3 de la OE son chequeadas por un empleado de depósito del centro de distribución, en caso de existir problemas las copias 1 y 3 son modificadas indicando el problema (mercadería en mal estado, exceso de mercadería, etc). Una vez que el camión está vacío, las copias 1 y 3 son firmadas en conformidad por el empleado del centro de distribución, quedando la copia 1 en el centro de distribución y la copia 2 y 3 regresan con el empleado de reparto. La copia 2 no se firma dado que su destino es el área administrativa, donde se la utiliza para liquidar las comisiones del empleado de reparto. La copia 3 retorna al área de logística como comprobante de la entrega exitosa de la mercadería. Cuando el empleado de reparto regresa la copia 4 de la OE es archivada permanentemente en el área de despacho.

A partir de este enunciado y con el uso de la herramienta *Coordinates Workbench* presentada en el Anexo A se modela el proceso descripto.

La herramienta representa las tareas como rectángulos con el nombre asignado, y los recursos como rectángulos con ángulos redondeados. Los arcos entre tareas y recursos representan las relaciones tarea-recurso, mientras que los arcos que unen dos tareas representan relaciones temporales. La figura 4.21 muestra el proceso de suministro para el área Santa Fe. Este proceso involucra las tareas: *Preparar Plan de distribución (PD), Imprimir orden de entrega (OE), Cargar harina industrial, Entrega exitosa, cerrar transacción de entrega*; todas estas tareas están vinculadas con la relación *Antesque*.

Sobre el lado izquierdo de la ventana del workbench, se representa la vista del dominio donde se muestran las tareas, recursos y versiones de las tareas que fueron

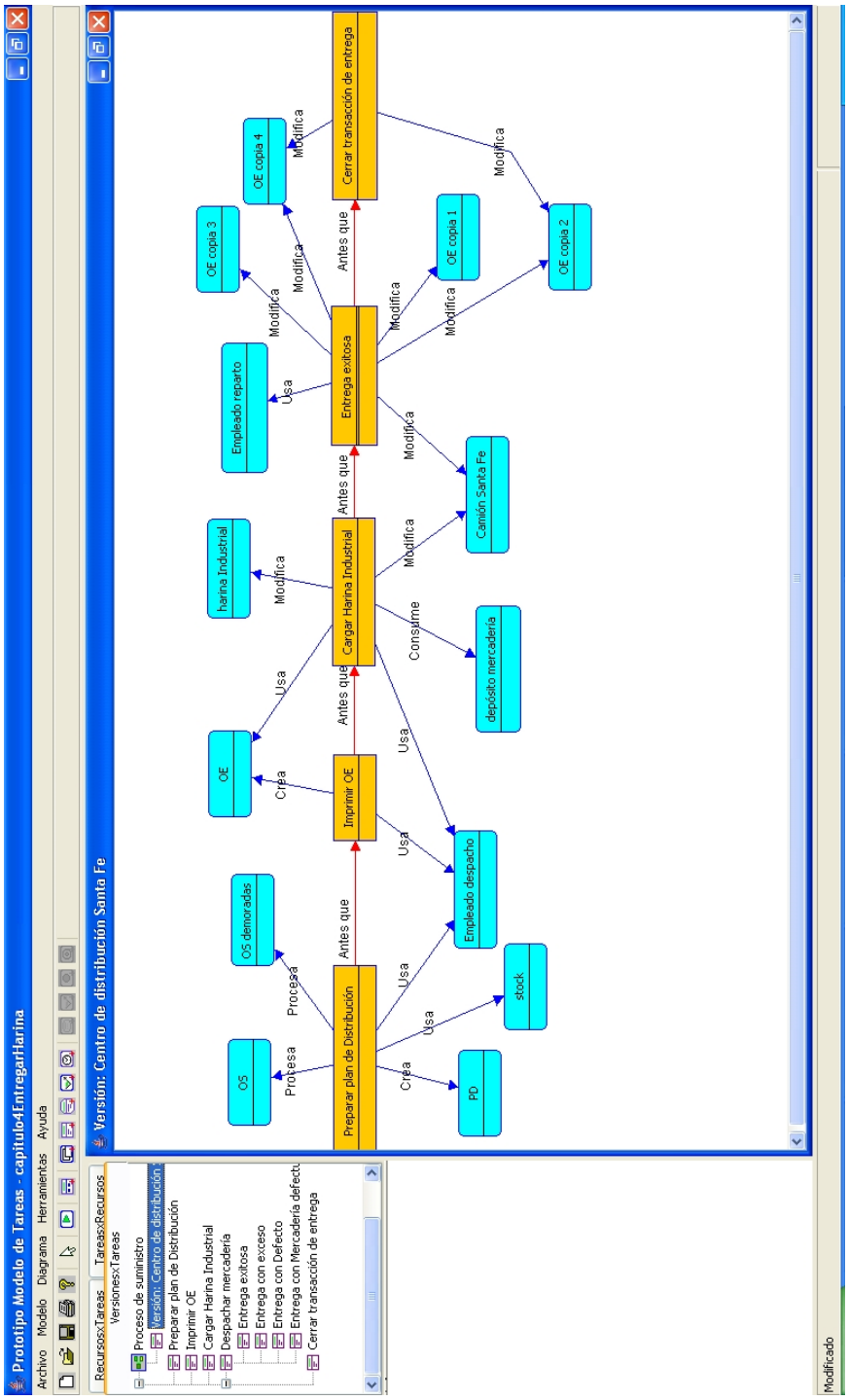


Figura 4.21: Proceso de Suministro

creadas.

Se puede ver que el proceso *Proceso de suministro* tiene una versión asociada llamada *Centro de distribución Santa Fe*, y la tarea *Despachar mercadería* tiene 4 versiones asociadas indicando las posibles formas de realizarse: (i) *Entrega exitosa* representa el caso cuando no existen inconvenientes en la entrega, (ii) *Entrega con exceso* cuando existe mercadería entregada en exceso y la misma es retornada, (iii) *Entrega con defecto* cuando falta mercadería por entregar y finalmente, (iv) *Entrega con mercadería defectuosa* representando el hecho que existe mercadería en mal estado que es devuelta por el centro de distribución.

Como se observa en la versión del proceso de suministro, la tarea *Despachar mercadería* aparece con el nombre de una de las versiones, en este caso *Entrega exitosa*, la cual se la identifica como la versión por defecto (default). El componente gráfico que la representa es diferente al de las otras tareas, como se observa, ésta presenta una doble línea debajo del nombre de la tarea indicando que la misma es una tarea compuesta y una vista más detallada es posible.

Así, la figura 4.22 muestra la versión por defecto asociada a dicha tarea, es decir *Entrega exitosa*. En la misma se presentan las tareas llevadas a cabo cuando la mercadería es entregada en el centro de distribución, y controlada por defectos posibles. En esta versión no se contempla la existencia de problemas en la entrega, luego, las órdenes de entrega son firmadas en conformidad por el empleado del centro de distribución encargado de recibir la mercadería.

De esta forma, las otras versiones de la tarea *Despachar mercadería* son desarrolladas indicando la existencia de problemas en la entrega, las cuales serán registradas en las copias de la orden de entrega.

Una vez que se finaliza con la descomposición de las tareas, y se representan todos los posibles casos de descomposición de las mismas, es posible establecer las relaciones

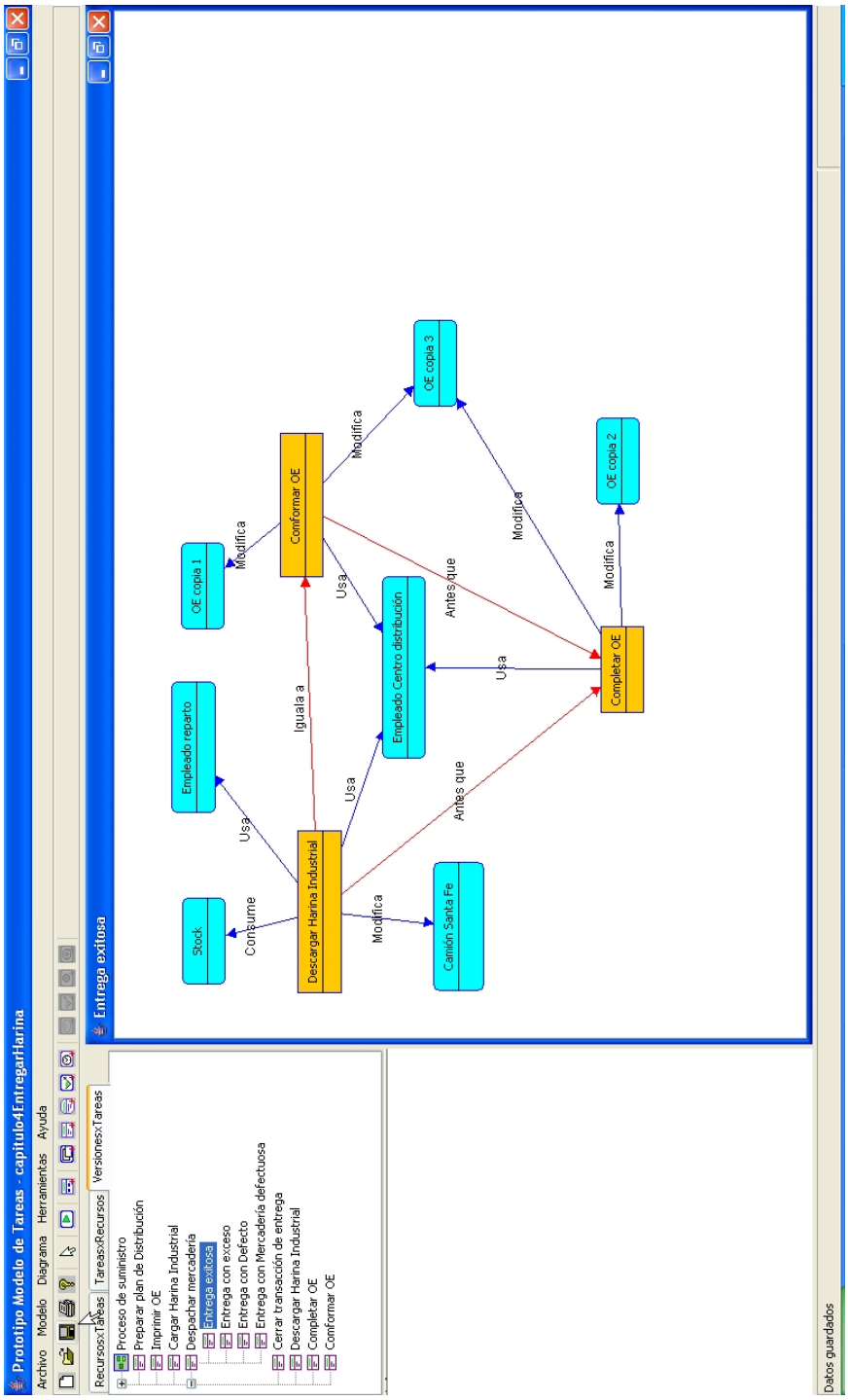


Figura 4.22: Versión de tarea: Entrega Exitosa

entre los recursos que interviene en la misma. Esto constituye la vista del dominio.

Como ejemplo se presenta el diagrama de recursos en la figura 4.23 donde aparece el recurso *orden de entrega* (OE) representado como un recurso compuesto cuyos componentes son sus cuatro copias.

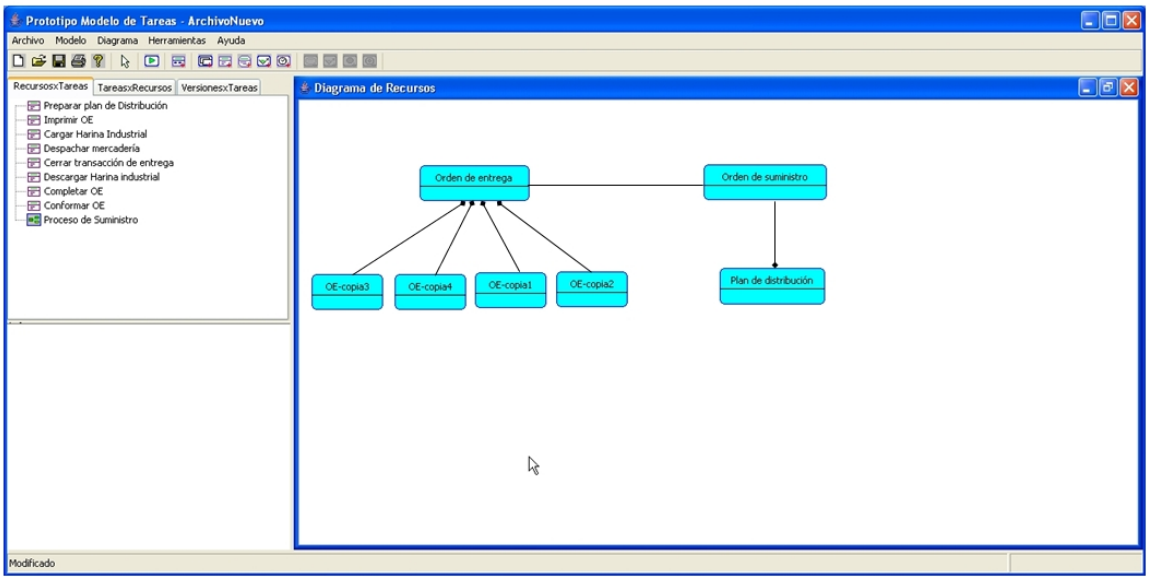


Figura 4.23: Diagrama de Recursos

Este diagrama muestra una vista parcial de la vista del dominio donde se incluyeron los recursos que intervienen en el proceso descripto.

En este diagrama de recursos, *plan de distribución* y *orden de suministro* aparecen vinculados, dado que un plan de distribución incluirá varias ordenes de suministro a ser cumplidas.

También existe una relación entre *orden de entrega* y *orden de suministro*, indicando que para cada orden de suministro del plan de entrega que se procesa, existirá una orden de entrega conteniendo información sobre la mercadería entregada, la cual deberá estar acorde con la mercadería solicitada.

Luego de haber representado en la vista del dominio las relaciones entre los recursos,

es necesario definir el ciclo de vida de los recursos. Para cada recurso que interviene, la vista dinámica muestra la evolución de los recursos como consecuencia de participar en las distintas tareas.

A modo de ejemplo se presenta en la figura 4.24 el ciclo de vida del plan de distribución. Este es creado semanalmente y usado para preparar las órdenes de entrega de esa semana. La tarea *Preparar plan de distribución*, cuando inicia su ejecución, crea el recurso *PD*, dejándolo en el estado *listo para procesar* cuando finaliza.

Este estado es el estado inicial para que la tarea *imprimir orden de entrega* pueda comenzar a ejecutar, esta tarea es realizada todos los días, por lo tanto, existe un ciclo en el diagrama del ciclo de vida del recurso entre los estados *procesando día* y *día procesado*.

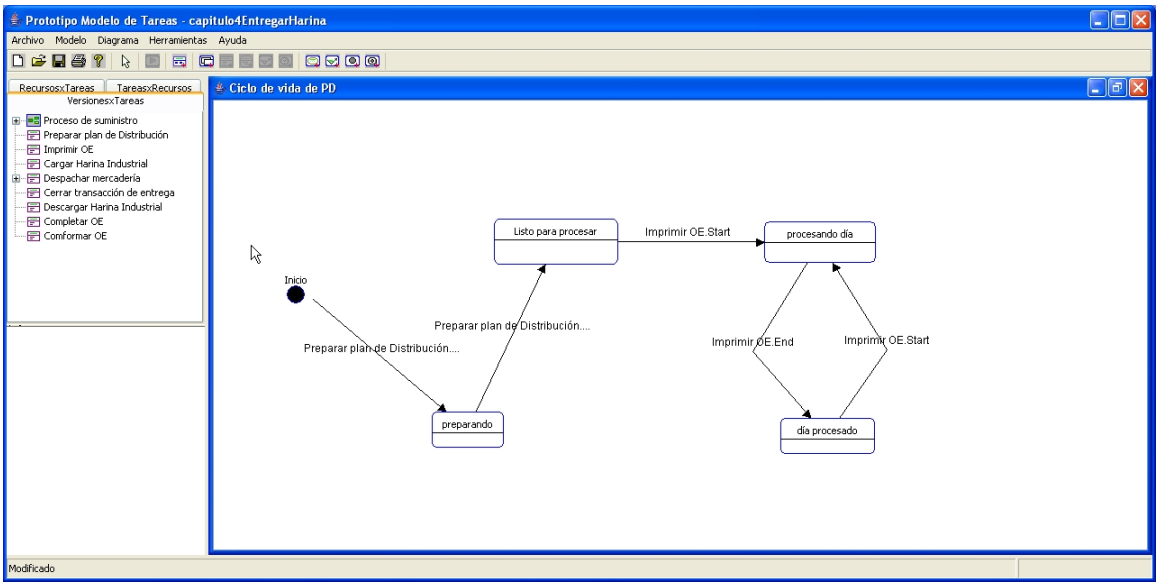


Figura 4.24: Ciclo de Vida del recurso PD

De esta forma todos los recursos tendrán su ciclo de vida, que identifican de que manera las tareas producen cambios de estados en los mismos.

Como se observa, el ejemplo muestra para un caso simple de un proceso de la

empresa, las distintas vistas que es necesario generar para obtener el modelo de la empresa.

Si bien la construcción requiere de un detalle considerable en la descripción de las tareas y recursos, el lenguaje utilizado facilita la labor dado que el mismo es un lenguaje amigable orientado a negocios, evitando incorporar componentes en el modelo que no son relevantes para el experto en procesos de negocios.

4.3. Conclusiones

En este capítulo se presentó el diseño de la capa de modelado conceptual perteneciente a la arquitectura propuesta. El diseño de la misma se orienta a proveer funcionalidades para la construcción del EM usando el lenguaje Coordinates. Se identificaron las tres vistas de modelado soportadas por la capa de modelado conceptual: vista de Tareas, vista del Dominio y vista Dinámica que conforman el modelo de empresa de acuerdo a lo establecido en Coordinates. La arquitectura de esta capa, está basada sobre el modelo *Document-View*, donde *Document* queda representada por el componente *Coordinates Model* de la arquitectura, el cual abstrae el conocimiento de la organización y *View* representado por el componente con el mismo nombre en la arquitectura, representa la interfaz gráfica que captura los eventos que el usuarios puede producir cuando está construyendo el modelo de empresa. Esta capa permite al modelador definir clases de tareas y de recursos a través de la componente *Abstraction* e instanciarlas para generar sus propios modelos de empresa. En esta capa, las tareas pueden ser representadas en diferentes niveles de abstracción haciendo uso del concepto de versión de tarea. Diferentes relaciones pueden ser usadas durante la construcción de los modelos de tareas, expresando no solo secuencias temporales sino también la manera en que una tarea usa o modifica a los recursos. Finalmente la vista dinámica permite analizar todas las tareas

que utilizan un recurso dado, así como también determinar cuáles son las consecuencias de compartir un recurso. Este capítulo intenta demostrar cómo la capa de modelado conceptual integra las diferentes vistas del modelo de empresa. El proceso iterativo e incremental que se propone para el desarrollo de estos modelos, está soportado por las flexibles funcionalidades que esta capa presenta. Se detalló la forma de llevar a cabo dichas funcionalidades describiendo diagramas de secuencias y diagramas de clases que dan soporte a las mismas. Por último se presentó un ejemplo para mostrar la manera en que esta capa de la arquitectura da soporte a la construcción de un EM a través de las diferentes vistas definidas por el lenguaje. En el desarrollo de este ejemplo se utilizó un prototipo de la arquitectura llamado *Coordinates workbench*.

5

Componentes del Modelo de Simulación

En este capítulo se aborda la descripción de los bloques de construcción del modelo de simulación. Éstos son modelo DEVS con interfaz y comportamiento bien definidos utilizados por la capa de simulación para la construcción del modelo de simulación.

Los componentes del modelo de simulación plasman el comportamiento de los conceptos encontrados en el modelo conceptual de la empresa. Los bloques de construcción utilizados para construir el Modelo de Empresa (EM) por Coordinates para sus modelos de tareas son: (i) *Tareas* (las cuales se clasifican en tarea atómica, tarea compuesta y proceso), (ii) *Recursos*, (iii) *Versión de Tarea* (representando una descomposición particular de una tarea compuesta), (iv) *relaciones tarea-recurso*, y (v) *relaciones temporales entre tareas*. A su vez, los recursos tienen sus propios diagramas, en los cuales intervienen los conceptos: (i) *estado*, (ii) *transición*, (iii) *evento*.

Las tareas atómicas, representan los bloques de construcción más simples dado que simbolizan actividades no descomponibles. Para su representación en el SM se propone utilizar modelos DEVS básicos (representado por la ecuación 2.3). Las tareas compuestas y procesos, por otro lado, representan estructuras complejas, que se describen como un conjunto de tareas más simples a través de la definición de versión de tarea. A estas estructuras se las asocia con modelos DEVS acoplados, los cuales tienen como compo-

nentes tanto modelos DEVS básicos como otros modelos acoplados. Los recursos, serán representados por modelos DEVS básicos, de esta manera, aquellos recursos compuestos serán representados independientemente un componente de otro. En este sentido, no habrá interacción entre los recursos directamente sino que la misma se producirá a través de las tareas en las cuales intervengan. Con respecto al ciclo de vida de los recursos, sus estados y transiciones se usan para definir las funciones de transición interna y externa del recurso y los eventos de salida generados por el modelo. Las relaciones temporales y relaciones tarea-recurso, se emplean para definir los acoplamientos entre los modelos DEVS correspondientes. La tabla 5.1 muestra las relaciones entre los bloques de construcción de los modelos de Coordinates y los bloques de construcción del modelo de simulación.

Bloques de construcción de modelos Coordinates	Modelos DEVS
Tarea atómica	Modelo DEVS básico
Versión de Tarea	Modelo DEVS acoplado
Recurso	Modelo DEVS básico
Relación tarea-recurso	Puertos de E/S y acoplamientos
Relaciones temporales	Puertos de E/S y acoplamientos
Estados de recursos	estados del modelo DEVS básico asociado al recurso
transición de estados del recurso	eventos de E/S

Tabla 5.1: Relación entre los componentes de un EM y un SM

Siguiendo estos lineamientos, los modelos DEVS se describen formalmente. Luego estos modelos se implementan extendiendo las clases provistas en el framework DEVS-JAVA. Se presenta a continuación la descripción formal de cada uno de los modelos que aparecen en la tabla 5.1 constituyendo los mismos, los bloques de desarrollo necesarios para generar el SM. Luego, la implementación de estos bloques y las reglas de transformación que serán presentadas más tarde, permitirán entender la estrategia de construcción de los modelos usando estos bloques de construcción.

Es necesario destacar, en el desarrollo de estos modelos, que los mismos deben representar el comportamiento de los componentes de un EM. La semántica asociada a estos componentes debe ser tenida en cuenta, dado que representa un aspecto importante en la definición de los componentes de un SM. Por ejemplo, cuando una relación tarea-recurso sea representada, la semántica asociada a ésta debe ser expresada con algún elemento del modelo de simulación DEVS. En otras situaciones, si una tarea *consume* un recurso,se indica que parte del recurso deja de existir, si una tarea *usa* un recurso significa que el estado del recurso cuando la tarea comienza y finaliza, debe ser el mismo, cuando una tarea *procesa* un recurso, significa que el recurso esperará en una cola ser atendido por la tarea, luego de lo cual, el recurso sufrirá un cambio de estado. Estos y los restantes conceptos incluidos en los EM expresados con el lenguaje Coordinates deberán ser traducidos e incorporados en los distintos componentes del SM expresado con el formalismo DEVS.

Los modelos DEVS que se definen para construir el SM son:

- AtomicTaskDevs que representa una tarea simple
- ResourceDevs que representa un recurso.
- TaskVersionDevs que representa la descomposición de una tarea en subtareas a través de la versión de tareas.
- SimulationModel que representa el proceso que se quiere ejecutar y es el modelo de mayor nivel.
- ExperimantalFrame que representa el modelo encargado de generar el experimento con el cual el modelo de simulación será analizado.

Como se puede observar en la lista dada anteriormente, se incluyeron dos modelos que hasta ahora no fueron nombrados, estos son: el modelo de simulación propiamente

dicho (SimulationModel), el cual representa el modelo de mayor nivel y el marco experimental (ExperimentalFrame) el cual representa el experimento con el que se evaluará el modelo de simulación. Este último modelo tiene como principal función, alimentar el modelo de simulación con eventos, determinar cuándo terminar la simulación y obtener los resultados de la misma.

A continuación se presentan cada uno de los modelos definidos, indicándose su equivalencia en el modelo Coordinates a partir del cual se originan.

5.1. AtomicTaskDevs

Este modelo DEVS representa la semántica del comportamiento de una tarea atómica que forma parte de un EM. Cada tarea atómica **T** que participa de un modelo conceptual, es representada en el modelo de simulación a través de un modelo *AtomicTaskDevs* que captura su semántica. La figura 5.1 representa gráficamente el modelo DEVS de una tarea atómica. Primeramente, se definen los puertos de entrada/salida que este modelo debe tener con el objetivo de recibir y procesar los eventos que envían otros modelos relacionados. Una tarea atómica en el modelo conceptual se relaciona con otras tareas a través de las relaciones temporales y con los recursos a través de las relaciones tarea-recursos. Por otro lado, este modelo puede recibir eventos que indiquen el final de la simulación.

Teniendo estas relaciones en cuenta se definen los siguientes puertos de entrada para este modelo:

ER : correspondiente a las entradas provenientes de los recursos. Por este puerto se recibirán los eventos que los recursos envíen.

ED : correspondiente a las entradas de instancias de recursos generados en forma ex-

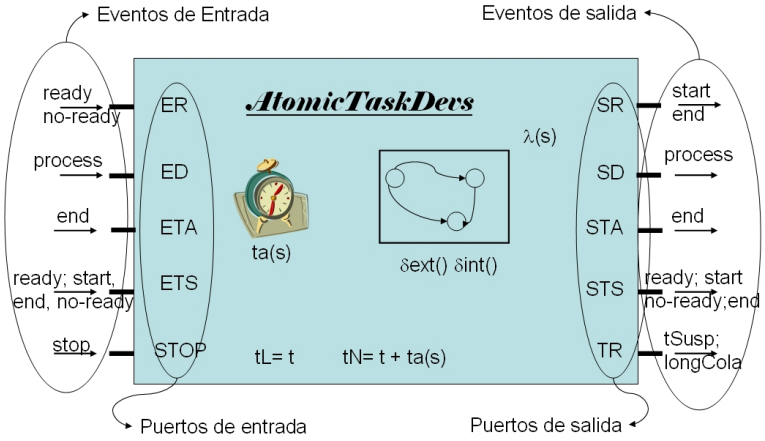


Figura 5.1: Diagrama del modelo DEVS que representa una tarea atómica

terna.

ETA : correspondiente a las entradas provenientes de tareas con relaciones temporales asíncronas.

ETS : correspondiente a las entradas provenientes de tareas con relaciones temporales síncronas.

STOP : correspondiente a la entrada proveniente del aceptor indicando fin de la simulación

Los puertos de salida fueron definidos teniendo en cuenta los modelos que están interesados en recibir eventos producidos por las tareas atómicas. Estos modelos son modelos relacionados a otras tareas (atómicas o compuestas) y a recursos, dado que los eventos de comienzo y finalización de una tarea causan transiciones de estados en los recursos participantes. También, la tarea atómica debe enviar eventos con las estadísticas que sirvan para obtener los resultados de una simulación. Luego, los puertos de salida que se definen son:

SR : correspondiente a las salidas dirigidas a los recursos relacionados.

SD : correspondiente a las salidas dirigidas a otras tareas que procesan el recurso enviado.

STA : correspondiente a las salidas dirigidas a tareas con relaciones temporales asíncronas.

STS : correspondiente a las salidas dirigidas a tareas con relaciones temporales síncronas.

TR : correspondiente a las salidas dirigidas al modelo encargado de obtener estadísticas.

Cada uno de los puertos de entrada tiene un conjunto de eventos definidos que pueden ser interpretados por el modelo. Cualquier otro evento recibido que no esté en el conjunto de eventos definidos, será considerado erróneo y descartado para procesar. De igual manera, los puertos de salida tienen asociados eventos que son generados por el modelo y cuyo destino final son puertos de entrada de otros modelos que estén acoplados. Por lo tanto debe existir una coherencia en la definición de los eventos de entrada/salida para los distintos modelos.

Para el caso de la tarea atómica, esquematizada en la figura 5.1, la tabla 5.2 muestra los posibles eventos para los puertos de entrada/salida definidos anteriormente.

Así, las tareas reciben desde los recursos, por medio del puerto ER, los eventos *ready* y *no-ready*, mediante los cuales se informa si el recurso está o no en el estado que es precondición de la tarea. En el caso de recursos que son procesados por las tareas, se reciben por medio del puerto ED, los eventos process(*instanciaRecurso*) donde *instanciaRecurso* corresponde a una instancia de recurso que debe ser procesado por la tarea o puesto en cola de espera si es que la tarea está ocupada procesando otro recurso.

Las tareas reciben los eventos *start* y *end*, por el puerto ETS, informando que la

Puertos de Entrada-salida	valores de eventos
ER	ready; no-ready
ED	process(instanciaRecurso)
ETS	start; end; ready; no-ready
ETA	end
STOP	stop
SR	start; end
SD	process(instanciaRecurso)
STS	start; end; ready; no-ready
STA	end
TR	tSusp; longCola

Tabla 5.2: Puertos de entrada/salida y eventos relacionados

tarea vinculada comienza o termina su ejecución. Por este mismo puerto se reciben los eventos *ready* y *no-ready* informando que la tarea vinculada está listas para comenzar a ejecutar. Estos eventos son utilizados para sincronizar los comienzos de las tareas cuando las relaciones temporales así lo requieren. En cuanto a los eventos de salida, la tarea debe informar a los recursos cuando comienza y termina su ejecución enviando por el puerto SR los eventos *start* y *end*. Estos eventos corresponden a los eventos *startTrigger* y *endTrigger* definidos en el modelo conceptual. Los eventos de comienzo y fin de la ejecución son enviados también por el puerto STS a las tareas relacionadas para sincronizar las mismas según sean las relaciones temporales. Para el caso de los recursos procesados, las tareas envían a otras tareas la instancia de recurso que procesó enviando el evento *process(instanciaRecurso)* por el puerto SD.

A partir de las definiciones realizadas se propone el modelo DEVS de tarea atómica a partir del modelo DEVS clásico (desarrollado en 2.1.2.2):

$$AtomicTaskDevs =< X_T, Y_T, \delta int_T, \delta_{ext_T}, S_T, ta_T, \lambda_T > \tag{5.1}$$

Cada uno de estos componentes se describen en los siguientes párrafos.

Conjunto de eventos de entrada ($\mathbf{X}_T$). Este conjunto describe los posibles valores que el modelo recibe por sus puertos de entrada. Para la tarea **T**, el conjunto de puertos

de entradas es:

$$ENTRADAS_T = \{ER_T, ED_T, ETS_T, ETA_T, STOP_T\} \tag{5.2}$$

El consiguiente conjunto de valores de eventos de entrada admitidos es expresado por medio del conjunto X_T:

$$\begin{aligned} X_T &= \{(ER_T, ready), (ER_T, no-ready), (ED_T, process(ir)), (ETS_T, ready), (ETA_T, end) \\ &\quad (ETS_T, no-ready), (ETS_T, start), (ETS_T, end), , (STOP_T, stop)\} \end{aligned} \tag{5.3}$$

Conjunto de eventos de salida ($\mathbf{Y}_T$) . Este conjunto define los posibles valores que genera el modelo y que están presentes en sus puertos de salida. Para la tarea T, el conjunto de puertos de salidas es:

$$SALIDAS_T = \{SR_T, SD_T, STS_T, STA_T, TR_T\} \tag{5.4}$$

El consiguiente conjunto de valores de eventos de salida es expresado por el conjunto Y_T:

$$\begin{aligned} Y_T &= \{(SR_T, start), (SR_T, end), (SD_T, process(ir), (STS_T, ready), (STS_T, no-ready), \\ &\quad (STS_T, start), (STS_T, end), (STA_T, end), (TR_T, tSusp), (TR_T, longCola)\} \end{aligned} \tag{5.5}$$

Una vez definidos los puertos de entrada y salida, y sus valores asociados, se definen los posibles estados en que una tarea atómica puede encontrarse. El diagrama de transición de estados de la figura 5.2 muestra los posibles estados de una tarea y sus transiciones de acuerdo a los eventos. Las transiciones que aparecen con línea llena corresponden a la función de transición de estado externa mientras que las transiciones de líneas punteada corresponden a la función de transición de estado interna.

Conjunto de estados ($\mathbf{S}_T$). Por lo dicho anteriormente, el conjunto de estados de la tarea **T** queda definido como:

$$S_T = \{init, executing, suspended, withoutRes, informing, waiting, suspWithoutRes\} \tag{5.6}$$

Para interpretar estos estados se analiza las funciones de transición de estados interna y externa para la tarea T.

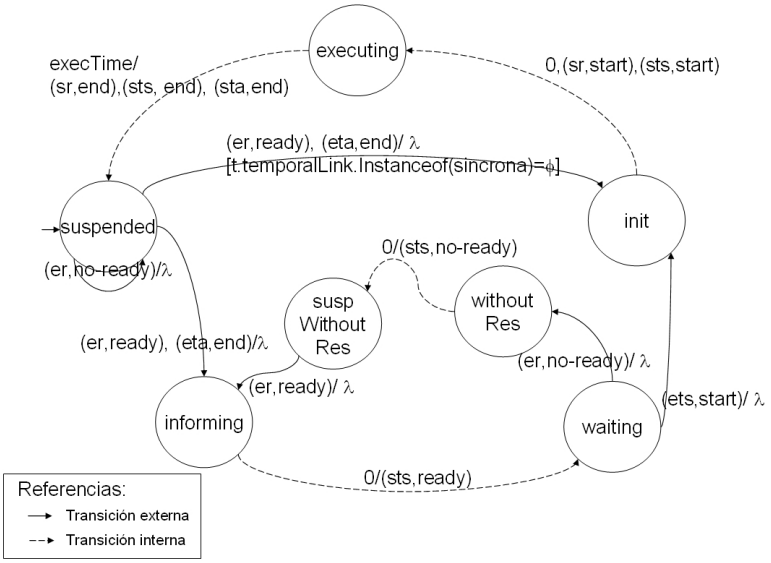


Figura 5.2: diagrama de transición de estado de AtomicTaskDevs

Función de transición de estado interna (δ_{int_T}). Esta función define cual es el próximo estado de la tarea **T** como consecuencia del paso del tiempo. La figura 5.2 muestra el comportamiento de una tarea, indicando sus transiciones internas y externas. Se comienza analizando el caso de una tarea **T** que se encuentra lista para ejecutar en el estado *init*, esto quiere decir que: (i) todos los recursos que utiliza están en los estados apropiados, (ii) todas las tareas que la preceden terminaron su ejecución y (iii) todas las tareas que deben comenzar junta con **T** están listas para ejecutar. En ese instante, **T** debe informar a todos sus recursos y a todas las tareas que deben empezar junto con **T**, que va a comenzar su ejecución, debiendo enviar para ello eventos que puedan ser interpretados como tales.

Cuando la tarea **T** está en el estado *init*, instantáneamente pasa al estado *executing*, produciendo como resultado eventos *start* que salen por los puertos SR_T y STS_T para informar a recursos y tareas relacionados que comenzó su ejecución. La siguiente transición se dará cuando transcurra un tiempo igual a la duración de la tarea **T** ($execTime$), en este caso la tarea que estaba en estado *executing* pasa a estar *suspended* hasta que

vuelva a encontrar las condiciones para poder ejecutar nuevamente. De igual manera que lo hizo cuando comenzó a ejecutar, **T** debe avisar a todas las tareas y recursos relacionados que finaliza su ejecución; esto lo hace generando eventos *end* que serán enviados por los puertos SR_T, STS_Y, STA_T. Las otras transiciones internas que se producen, tienen como objetivo el envío de eventos externos. Cuando **T** está en el estado *informing*, pasa a un estado *waiting* y emite el evento de salida *ready* por el puerto STS_T. Puede suceder que estando en el estado *waiting*, los recursos cambien de estado y envíen un evento *no-ready*, en cuyo caso, la tarea deja de estar lista para ejecutar, y debe entrar en el estado *suspWithoutRes* (estado que representa la condición de suspendida por no estar el recurso disponible) informando a las tareas que deja de estar lista enviando el evento *no-ready* por el puerto de salida STS_T. Para poder generar este evento externo, se agrega un estado *withoutRes* para que la Tarea genere dicho evento y pase al estado *suspWithoutRes*, donde queda esperando a recibir las condiciones de ejecución de los recursos.

Función de transición de estado externa (δ_{ext_T}). Esta función define cual es el siguiente estado de **T** cuando la misma recibe un evento externo, la figura 5.2 muestra junto con las transiciones internas (líneas punteadas), las transiciones externas (líneas llenas). Inicialmente la tarea **T** está en el estado *suspended* hasta que pueda empezar a ejecutar (es decir encuentre todas las condiciones), para ello debe cumplirse que:

- Recibe eventos *start* en el puerto de entrada ETS_T proveniente de todas las tareas con relaciones síncronas.
- Recibe eventos *end* en el puerto de entrada ETA_T proveniente de todas las tareas con relaciones asíncronas.
- Recibe eventos *ready* en el puerto de entrada ER_T proveniente de todos los re-

cursos que utiliza.

Cuando la tarea T está suspendida y recibe los eventos *ready* por su puerto ER_T y *end* en el puerto ETA_T, y no es destino de relaciones temporales síncronas, la misma pasa al estado *init* (lista para ejecutar). Como se dijo anteriormente, no es posible emitir ningún evento de salida por lo que se indica en la transición que la salida es λ. El otro escenario posible es que la tarea T sea destino de relaciones temporales síncronas, en cuyo caso la transición que se ejecuta es la transición que deja a la tarea en el estado *informing*, de manera que la tarea pueda informar a las otras, que está lista para ejecutar. Cuando la tarea T está esperando que otras tareas encuentren las condiciones para ejecutarse en el estado *waiting*, puede recibir el evento *no-ready* de algún recurso relacionado, con lo cual pierde sus condiciones para ejecutar y debe regresar a estar *withoutRes* hasta encontrar de nuevo las condiciones de los mismos. En otro caso, estando en este estado, la tarea recibe los eventos *start* de las otras tareas relacionadas y pasa al estado *init* para comenzar a ejecutar.

Función del tiempo ($\mathbf{ta}_T$). Esta función indica cuanto tiempo la tarea está en cada uno de los estados. Para el modelo presentado ta_T queda definido como:

ta_T(init)= 0

ta_T(suspWithoutRes)=∞

ta_T(executing)= execTime

ta_T(withoutRes)= 0

ta_T(suspended)= ∞

ta_T(waiting)= ∞

ta_T(informing)= 0

Los estados de duración cero, son estados ficticios necesarios para generar eventos de salida. Los estados de duración ∞ son estados permanentes, la única forma de

abandonarlos es a través de la recepción de eventos externos.

Función de salida (λ_T). Esta función define los eventos generados por el modelo cuando se producen transiciones internas. Determina cual es el evento y por que puerto se emite, dependiendo del estado en que se encuentre. Entonces, la función de salida para un modelo asociado a la tarea T queda definida de la siguiente forma:

λ_T(init)={y(STS$_T$, ready), y(SR$_T$, start)}

λ_T(executing)={y(SR$_T$, end), y(STA$_T$, end)}

λ_T(informing)= {y(STS$_T$,ready)}

λ_T(withoutRes)= {y(STS$_T$,no-ready)}

5.2. ResourceDevs

Cuando se analizan los recursos, desde el punto de vista de su comportamiento, se observa que el mismo queda definido por su DTE asociado, el cual representa la evolución del recurso como consecuencia de participar en las tareas. Sin embargo no existe una relación directa entre recursos de manera que uno afecte o provoque un cambio de estado en otro. Esta situación no está representada en el modelo conceptual. Si bien, en los DTE asociados a recursos existe el concepto de super-estado, sub-estado y estados ortogonales, estos tienen un significado especial. Los super-estados y sub-estados, representan los diferentes estados que puede atravesar un recurso cuando participa de versiones de tareas. De esta manera se puede representar un super-estado indicando el estado en que el recurso se encuentra al participar en una versión, mientras que los sub-estados correspondientes reflejan los estados del recurso al participar en las tareas atómicas involucradas en la versión. Los estados ortogonales reflejan la existencia de recursos compuestos donde cada estado ortogonal refleja las transiciones de estado o

comportamiento de cada una de las partes que integran el recurso. Así cada componente evoluciona independientemente del otro a medida que participan de las tareas.

A partir de este análisis se puede decir que es posible representar un recurso con un comportamiento simple, constituido por estados y transiciones causadas por la ejecución de las tareas atómicas de las cuales participa. Se optó utilizar un modelo DEVS atómico para represente el comportamiento del recurso como se muestra en la figura 5.3.

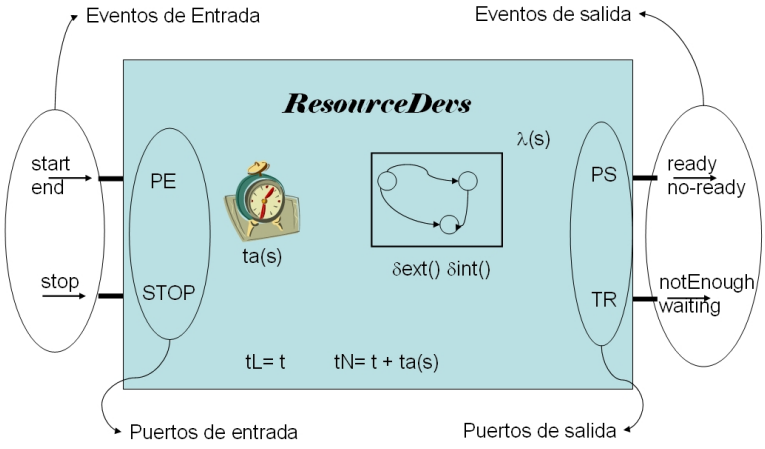


Figura 5.3: Diagrama del modelo ResourceDevs

El modelo *ResourceDevs* queda definido por la expresión 5.7:

$$ResourceDevs =< X_R, Y_R, \delta_{int_R}, S_R, \delta_{ext_R}, \lambda_R, ta_R > \tag{5.7}$$

Para poder definir sus puertos de entrada y salida, se analizan las relaciones establecidas entre un recurso y una tarea: por un lado, las tareas causan cambios de estados en los recursos, y por otro lado la ejecución de una tarea depende del estado en que se encuentre el recurso que utiliza. Un recurso **R** tiene asociado un DTE_R que describe su ciclo de vida (figura 5.4). En el mismo, se definen estados y transiciones, estas últimas están relacionadas con Tareas a través de dos relaciones: *startTrigger* y *endTrigger* que identifican las tareas que al comenzar o terminar su ejecución causan la transición de

estado en el DTE_R. Consecuentemente, el recurso debe avisar a la tarea cuando están dadas las condiciones para que la misma ejecute, y por otro lado, la tarea debe avisar al recurso cuando comienza o termina su ejecución de manera que éste puede cambiar su estado.

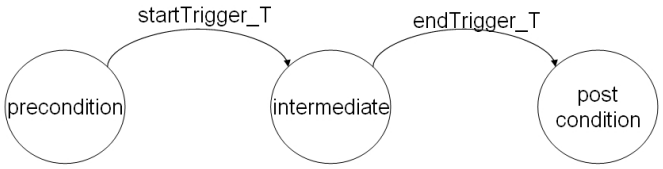


Figura 5.4: Diagrama de Transición de estado de un Recurso

Un modelo DEVS que represente el comportamiento de un recurso, tendrá un puerto de entrada por donde recibe los eventos de las tareas que lo usan y uno de salida por donde envía, a las tareas relacionadas, los eventos. Al igual que con el modelo de tareas, este modelo DEVS tendrá un puerto *STOP* por donde recibe el evento de fin de simulación y un puerto *TR* por donde envía los valores que serán usados para calcular las estadísticas que conforman los resultados de la simulación. Los conjuntos ENTRADAS_R y SALIDAS_R se definen como sigue:

$$ENTRADAS_R = \{PE_R, STOP_R\} \tag{5.8}$$

$$SALIDAS_R = \{PS_R, TR_R\} \tag{5.9}$$

Los posibles valores que pueden aparecer en estos puertos se encuentran definidos en la tabla 5.3. Así una tarea debe informar que comienza o termina su ejecución con los eventos *start* y *end*, y un recurso debe indicar a la tarea **T** que el estado actual del recurso corresponde a un estado inicio de la tarea, enviando el evento *ready* por el puerto de salida correspondiente. En caso de que el recurso no haya recibido el evento *start* de esa tarea y el estado del mismo cambie como consecuencia de otra tarea, el recurso debe enviar el evento *no-ready* a la misma.

Puertos de Entrada-salida	valores de eventos
PE_R	start; end
$STOP_R$	stop
PS_R	ready; no-ready
TR_R	notEnough; waiting

Tabla 5.3: Puertos de E/S para ResouceDevs

A partir de este análisis se describen a continuación los componentes del modelo *ResourceDevs*.

Conjunto de eventos de entrada ($\mathbf{X}_R$). El conjunto de eventos de entrada queda definido por todos aquellos eventos que pueden ser interpretados correctamente por el modelo DEVS *ResourceDevs*. Según fue explicado anteriormente éste se define como:

$$X_R = \{(PE_R, start), (PE_R, end), (STOP_R, stop)\} \tag{5.10}$$

Por el puerto de entrada, el recurso **R** recibe eventos *start* y *end*. Con dichos eventos las tareas informan al recurso que comienzan o terminan su ejecución. Estos eventos causan transiciones de estado en el recurso y son los eventos equivalentes a *startTrigger* y *endTrigger* definidos por el lenguaje Coordinates.

Conjunto de eventos de salidas ($\mathbf{Y}_R$). Este conjunto queda definido por todos aquellos eventos que pueden ser interpretados correctamente por modelos DEVS conectados a *ResourceDevs*. El mismo queda definido como:

$$Y_R = \{(PS_R, ready), (PS_R, no - ready), (TR_R, notEnough), (TR_R, waiting)\} \tag{5.11}$$

El recurso **R**, envía el evento *ready* por los puertos de salida conectados a puertos de entradas de modelos asociados a tareas que requieren el recurso **R** en el estado actual, y el evento *no-ready* a todas aquellas tareas que no lo requieren en ese estado. Para ello el modelo debe conocer cual es el estado del recurso que es precondición para cada tarea, dicha información se obtiene del DTE asociado al recurso en el modelo conceptual.

Conjunto de estados (S_R). Los estados de un recurso están definidos en el DTE del modelo conceptual. Por lo tanto este diagrama está representando los estados y las transiciones sufridas por el recurso como consecuencia de los eventos *startTrigger* y *endTrigger*. Al incorporar estos estados y transiciones en un modelo DEVS equivalente se observa que el conjunto de estados atómicos del DTE (llamado S_{DTE}) del modelo conceptual es un subconjunto del conjunto de estados del modelo DEVS (llamado S_R) que representa su comportamiento:

$$S_{DTE} \subset S_R \tag{5.12}$$

Esta afirmación es correcta si se analiza que todas las transiciones provocadas por el inicio o la finalización de la ejecución de tareas atómicas, que aparecen en el DTE del modelo conceptual se corresponden a transiciones externas en el modelo DEVS, esto quiere decir, transiciones causadas por eventos externos recibidos por los puertos de entrada (eventos *start* y *end* en los puertos de entrada del modelo *ResourceDevs*). Luego, se necesita definir las transiciones internas para este modelo, es decir aquellas causadas por el paso del tiempo. Estas transiciones son necesarias dado que sin ellas sería imposible que un modelo *ResourceDevs* genere eventos de salida. Fundamentalmente, el recurso necesita generar los eventos externos para poder informar su estado a las tareas relacionadas. Consecuentemente, hace falta incorporar nuevos estados, referenciados como estados *dummy*, para representar las transiciones internas y la generación de los eventos de salida. Por lo tanto el conjunto de estados de un *ResourceDevs* estará dado por el siguiente conjunto:

$$S_R = S_{DTE} \cup DUMMY \tag{5.13}$$

donde DUMMY es el conjunto de estados *dummy* agregados.

Función de transición de estado interna (δ_{int_R}). El DTE asociado a un recurso en el modelo conceptual es un modelo del estilo *event-driven*, es decir las transiciones son generadas por eventos externos. Pero ¿que sucede con los eventos internos? Estos no se encuentran definidos. Considérese una porción de un DTE como el que se muestra en la figura 5.4 que identifica en líneas generales de qué manera el mismo se encuentra

definido. Se observa que de un estado *precondition* se pasa a un estado *intermediate* por el evento *startTrigger* correspondiente a una tarea $T(\text{startTrigger}_T)$. Luego, desde este estado *intermediate* se pasa al estado *postcondition* cuando el evento *endTrigger* sucede. Este formato se repite para el conjunto de tareas que usan al recurso: cada una de ellas hace evolucionar al recurso de un estado a otro. El estado *precondition* corresponde al estado del recurso que es precondición para la tarea T y el estado *postcondition* corresponde al estado final en que la tarea deja el recurso una vez que ésta se ejecutó, es decir su postcondición. Ahora bien, para poder definir correctamente dónde incorporar las transiciones internas es necesario responder la pregunta: ¿en que momento el recurso debe enviar eventos hacia otros modelos?

La respuesta es simple: Antes de pasar a un estado marcado como *precondition* para una tarea, el recurso debe informar de dicha transición a la tarea interesada y luego pasar a ese estado. Considerando que el estado *precondition* de una tarea es el estado *postcondition* para otra, es posible establecer que antes de pasar al estado *postcondition* cuando se recibe el evento *endTrigger* debe producirse una transición interna con el fin de enviar el evento externo hacia las tareas cuya precondición coincida con el estado *postcondition* que se está por alcanzar.

Introduciendo esta modificación en el diagrama del recurso, el mismo puede verse como el que aparece en la figura 5.5 donde el estado coloreado *dummy* se incorpora como destino de la transición rotulada *endTrigger*. Una transición interna se agrega teniendo como origen el estado *dummy* y como destino el estado *postcondition* de la transición original. El rótulo asociado a la transición interna puede leerse como: evento/salida. En este caso, el cero identifica al evento simbolizando que transcurrió cero tiempo y la salida asociada a la transición es el evento *ready* que será enviado por el puerto *ps*.

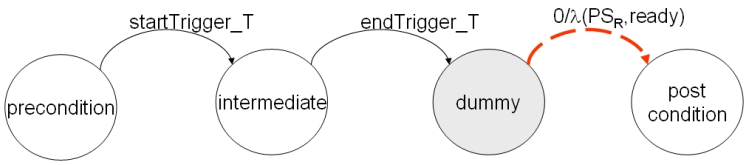


Figura 5.5: Incorporación de la transición interna

Función de transición de estado Externa (δ_{ext_R}). La función de transición de estado externa, queda definida por el DTE_R asociado al recurso **R** en Coordinates, ya que en el mismo se especifican todas las transiciones de estados de **R** causadas por la ejecución de las tareas. Es necesario sin embargo, modificarlo para poder representar los eventos externos como pares de la forma (puerto, evento) tal que este par esté comprendido en el conjunto X_R. En el DTE_R los eventos definidos son: $startTrigger_T$ y $endTrigger_T$ los cuales tienen referencia a la tarea asociada que genera el evento. Luego, esta representación debe transformarse para que pueda ser incorporada a un modelo DEVS. De esta manera, cada vez que un evento $startTrigger_{tareai}$ esté definido en DTE_R, se interpreta como la ocurrencia de un evento *start* en el puerto de entrada PE_R. De igual manera, la existencia de una transición rotulada con el evento $endTrigger_{tareai}$ será interpretada como la ocurrencia de un evento *end* en el puerto de entrada PE_R. El diagrama de la figura 5.6 muestra esta transformación.

Función del tiempo (ta_R). Como se indicó previamente, el recurso **R** tiene dos conjuntos de estados: (i) conjunto S_{DTE} de estados definidos en el DTE_R y (ii) conjunto $DUMMY$ de estados ficticios agregados para generar eventos externos. La función del tiempo queda definida como:

$\text{ta}_R(\text{q}) = \infty \ \forall \ \text{q} \ \epsilon \ \text{S}_{DTE}$

$\text{ta}_R(\text{q}) = 0 \ \forall \ \text{q} \ \epsilon \ DUMMY$

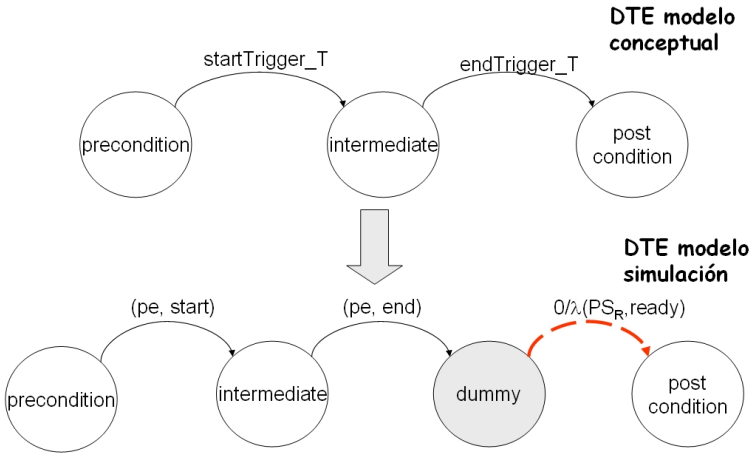


Figura 5.6: Transición Externa de un ResourceDevs

Función de salida (λ_R). Esta función indica cuál es el evento producido por el modelo *ResourceDevs* cuando se genera una transición interna. Estas transiciones se producen cuando el modelo pasa de un estado *dummy* a otro estado. Estos estados generan eventos *ready* y *no-ready* por los puertos de salidas PS_R. La siguiente es una descripción genérica de esta función:

λ_R(q)= {(PS_R,ready), (PS_R,no-ready), (TR_R, waiting)(TR_R, notEnough)}

5.3. TaskVersionDevs

Este modelo es definido para representar el comportamiento de una tarea compuesta o tarea proceso que tiene versiones de tareas asociadas. Dichas versiones representan la descomposición de una tarea en tareas más simples, con lo cual su comportamiento es el resultado del comportamiento de sus componentes. Dada una tarea compuesta **V** que participa de un modelo conceptual, la misma tiene asociada una Version de Tarea a través de su vínculo *default*. Para representar esta tarea en el modelo de simulación, se emplea un modelo DEVS acoplado (definido en la sección 2.1.2.2 por la expresión 2.3) llamado $TaskVersionDevs_V$ que representa el comportamiento de la tarea V a través de su versión asociada *default*. Este modelo, expresado en la ecuación 5.14, representa

también una tarea, al igual que *AtomicTaskDevs*. Consecuentemente, ambos modelos deben tener la misma interfaz, de manera de manipular ambos sin necesidad de distinguir uno de otro. Luego, los puertos de entrada y salida como así los valores presentes en dichos puertos son idénticos a los presentados para el modelo *AtomicTaskDevs*. Por otro lado, los componentes del modelo *TaskVersionDevs* están determinados de acuerdo a los componentes de la versión *default* en el modelo conceptual. Esto quiere decir que si la versión *default* de la tarea V tiene como componente una tarea atómica T, luego su modelo de simulación asociado, $AtomicTaskDevs_T$, formará parte del modelo acoplado $TaskVersionDevs_V$. Los recursos que participan de una versión no formarán parte del modelo acoplado $TaskVersionDevs_V$, dado que estos serán agrupados en el modelo de mayor nivel correspondiente a la tarea proceso que está asociada al modelo *SimulationModel*.

$$TaskVersionDevs =< X_V; Y_V; D_V; \{M_i\}; EIC_V; EOC_V; IC_V > \tag{5.14}$$

Por lo dicho anteriormente, se describen a continuación, los elementos de la tupla del modelo DEVS acoplado $TaskVersionDevs_V$ (expresión 5.14).

Conjunto de eventos de entrada ($\mathbf{X}_V$). Este conjunto describe los posibles valores que el modelo recibe por sus puertos de entrada. Como fue explicado, este modelo acoplado tiene el mismo conjunto de puertos de entrada y salida y los mismos valores asociados que los definidos para *AtomicTaskDevs* (tabla 5.2). De esta manera, una tarea atómica y una tarea compuesta tendrán la misma interfaz en el modelo de simulación. Se define al conjunto de puertos de entradas de la tarea **V** como:

$$ENTRADAS_V = \{ER_V, ED_V, ETS_V, ETA_V, STOP_V\} \tag{5.15}$$

El conjunto de eventos de entrada para dichos puertos queda definido como:

$$\begin{aligned} X_V \quad = \quad & \{(ER_V, ready), (ER_V, no-ready), (ED_V, process(ir)), (ETS_V, ready), \\ & (ETS_V, no-ready), (ETS_V, start), (ETA_V, end)\} \end{aligned} \tag{5.16}$$

Conjunto de eventos de salida ($\mathbf{Y}_V$). Esta definición es idéntica a la especificada para modelos atómicos. Este conjunto define los posibles valores que genera el modelo

y que están presentes en sus puertos de salida. El conjunto de puertos de salidas queda definido como:

$$SALIDAS_V = \{SR_V, SD_V, STS_V, STA_V, TR_V\} \tag{5.17}$$

y el conjunto de eventos de salida como:

$$\begin{aligned} Y_V \quad = \quad & \{(SR_V, start), (SR_V, end), (SD_V, process(ir), (STS_V, ready), (STA_V, end), \\ & (STS_V, no-ready), (STS_V, start), (TR_V, tSusp)(TR_V, longCola)\} \end{aligned} \tag{5.18}$$

Conjunto de referencia a sus componentes ($\mathbf{D}_V$). Este conjunto contiene las referencias a los modelos DEVS básicos o acoplados asociados a las tareas que intervienen en la versión de tarea bajo consideración. Para el caso particular de los recursos, estos no formarán parte del modelo acoplado que se define, dado que un recurso puede estar siendo usado por más de una versión de tarea. En este caso, los modelos DEVS de los recursos estarán conectados a través de sus puertos con los modelos que representan las tarea que los usan y formarán parte del modelo de mayor nivel o *SimulationModel*.

Conjunto de modelos DEVS componentes ($\{\mathbf{M}_i\}$). A partir de las referencias pertenecientes al conjunto D_V definido anteriormente, es posible acceder a las definiciones de cada uno de los modelos componentes:

$$\forall\, i \,\epsilon\, D_V,\ M_i \ = \ \{X_i, Y_i, S_i, \delta_{int_i}, \delta_{ext_i}, \lambda_i, ta_i\} \tag{5.19}$$

Acoplamiento interno ($\mathbf{IC}_V$). El acoplamiento interno (Internal Coupling) representa la manera en que los componentes del modelo $TaskVersionDevs_V$ están acoplados unos con otros. De esta manera IC_V es el conjunto de pares (puerto de salida / puerto de entrada) de los correspondientes componentes conectados a través de éstos. Dado que los componentes representan tareas en el modelo conceptual, un modelo estará acoplado con otro (es decir enviará eventos a otro modelo) cuando existan relaciones temporales que conecten las tareas en el modelo conceptual. Dependiendo de este tipo de relaciones los distintos puertos de salida de un componente estarán relacionados

con puertos de entrada de otros componentes. Así, cada vez que una tarea T_1 esté conectada con una relación temporal síncrona con otra tarea T_2, y siendo los modelos DEVS asociados en el modelo de simulación DT_1 y DT_2 respectivamente entonces se cumplirá que:

$$IC_V \supset \{((DT_1, STS_{DT1})(DT_2, ETS_{DT2}))\} con DT_1, DT_2 \epsilon D_V \tag{5.20}$$

De la misma manera, sucede con las relaciones temporales asíncronas. Luego, si existe una relación temporal asíncrona entre T_1 y T_2 se cumplirá que:

$$\begin{aligned} IC_V \supset\ & \{((DT_1, STA_{DT1})(DT_2, ETA_{DT2}))\} \cup \{((DT_1, SD_{DT1}), (DT_2, ED_{DT2}))\} \\ & con\ DT_1, DT_2 \epsilon D_V \end{aligned} \tag{5.21}$$

Para el caso de relaciones asíncronas los puertos *SD* y *ED* estarán acoplados también, indicando qué tarea le sigue en el procesamiento de un recurso dado.

Acoplamiento Externo de entrada ($\mathbf{EIC}_V$). Esta relación describe la manera en que el modelo acoplado delega en sus componentes los eventos externos que recibe. El conjunto EIC_V (de su sigla en inglés External Input Coupling) determina el acoplamiento existente entre los puertos de entrada del modelo acoplado y los puertos de entrada de los modelos componentes. Se pueden identificar tres tipos de eventos en la entrada que deben ser tratados en forma diferentes: (i) eventos que provienen de modelos *ResourceDevs*, los cuales deben ser propagados a todos los componentes a través de los puertos correspondientes; (ii) eventos que envían otras tareas (ya sean estas modelos *AtomicTaskDevs* o *TaskVersionDevs*) en cuyo caso sólo algunos componentes estarán interesados en recibir dichos eventos; (iii) eventos de fin de simulación que deben ser propagados a todos los componentes para que finalicen la simulación.

El primer grupo de estos eventos, los que envían modelos *ResourceDevs*, generan acoplamientos entre los puertos *ER* del modelo $TaskVersionDevs_V$ y los puertos *ER* de

todos sus componentes. De esta manera se garantiza que las tareas conocen siempre el estado de los recursos, aún cuando estos pertenezcan a un modelo superior en la jerarquía. Luego es posible identificar el primer grupo de acoplamientos que forman el EIC_V como sigue:

$$EIC_V \supset \{((V, ER_V), (d, ER_d))\} \cup \{((V, ED_V), (d, ED_d))\} \; \forall \; d \; \epsilon \; D_V \tag{5.22}$$

Esto indica que los puertos de entrada ER_V y ED_V del modelo acoplado $TaskVersionDevs_V$ están conectado a los puertos de entrada ER_d y ED_d de cada uno de los modelos d componentes.

El segundo grupo de eventos, hace referencia a los eventos recibidos por los puertos ETA_V y ETS_V. Estos eventos indican el comienzo o fin de la ejecución de tareas que preceden a la tarea $TaskVersionDevs_V$. Si se analiza esta situación, se puede entender que el único componente que está interesado en recibir estos eventos es el modelo que corresponde a la tarea que debe ejecutarse primera en la descomposición de $TaskVersionDevs_V$. Ahora bien, esta primer tarea, es aquella que cumple la condición de no ser destino de ninguna relación temporal *antes-que*. Esta condición establece que no existe otra tarea en la versión que deba ejecutarse antes, por lo tanto es la primera. Sin embargo en el modelo $TaskVersionDevs_V$ no aparece el concepto de relación temporal, este concepto solo se encuentra en el modelo conceptual. Es necesario entonces poder identificarla a través de acoplamientos que se producen entre los modelos de las tareas que integran el modelo acoplado $TaskVersionDevs_V$. De esta forma, si una tarea t_i es un componente de $TaskVersionDevs_V$ y su puerto de entrada ETA_{ti} no se encuentra acoplado a ningún puerto STA_{tj} de alguna tarea t_j perteneciente a $TaskVersionDevs_V$, luego se puede afirmar que t_i es la primera tarea en la versión. Si se analiza la versión, es posible descubrir que no necesariamente esta condición sea cumplida por una única tarea, sino que es posible encontrar varias tareas que cumplan con esta restricción. Este podría ser el caso de tareas que se ejecutan en paralelo. Con lo dicho se puede definir el conjunto *firstTask* de todas las tareas que deben ejecutarse inicialmente como sigue:

$$firstTask = \{d/d \; \epsilon \; D_V \wedge \nexists \, t_j \; \epsilon \; D_V/((t_j, STA_{tj}), (d, ETA_d)) \; \in IC_V\} \tag{5.23}$$

Habiendo identificado estas tareas, es posible definir el acoplamiento entre los puertos ETA_V y ETS_V de $TaskVersionDevs_V$ con los puertos ETA_d y ETS_d de cada modelo d que pertenece al conjunto *firstTask*. Esto queda expresado como sigue:

$$EIC_V \quad \supset \quad ((V, ETS_V), (d, ETS_d)) \cup ((V, ETA_V), (d, ETA_d)) \forall\, d \,\epsilon\, firstTask \tag{5.24}$$

Finalmente se analizan los acoplamientos que causan los eventos recibidos por el puerto $STOP_V$ correspondiente a fin de simulación. Es necesario que este evento sea propagado a todos los componentes de manera de finalizar la simulación en dicho componente. Luego, el puerto $STOP_V$ de $TaskVesionDevs_V$ está acoplado a los puertos $STOP_d$ de cada componente d perteneciente a $TaskVersionDevs_V$, lo cual puede expresarse como sigue:

$$EIC_V \quad \supset \quad \{((V, STOP_V), (d, STOP_d))\} \forall\, d \,\epsilon\, D_V \tag{5.25}$$

Acoplamiento Externo de Salida ($\mathbf{EOC}_V$). Este conjunto determina la relación que existe entre los puertos de salidas de los modelos componentes de un modelo acoplado con los puertos de salida del mismo. De esta manera queda establecida la forma en que los eventos de salida producidos por los componentes son propagados hacia otros modelos fuera del modelo acoplado. Como se realizó para la definición de EIC_V aquí también se identifican distintos tipos de eventos que deben ser analizados por separados. Estos son: (i) eventos de salidas en los puertos SR, los cuales tienen como destino los recursos, por lo tanto deben ser propagados hacia arriba en la jerarquía de descomposición; (ii) eventos que tienen como destino otras tareas fuera del modelo componente; (iii) eventos con valores estadísticos.

En el primero de estos grupos se puede establecer que los puertos de salidas SR_d de los componentes d, deben estar conectados al puerto de salida SR_V de $TaskVersionDevs_V$. Esto queda expresado en la siguiente expresión:

$$EOC_V \quad \supset \quad \{((d, SR_d)(V, SR_V))\} \,\cup\, \{((d, SD_d)(V, SD_V))\} \,\forall\, d \,\epsilon\, D_V \tag{5.26}$$

Para el segundo grupo de eventos, los que tienen como destinos otras tareas, se puede notar que no todos los modelos componentes envían eventos a otras tareas fuera de la versión. Principalmente se observa que las tareas que se ejecutan últimas en la versión son las que generarán los eventos de salidas hacia otras tareas que se encuentran fuera de la versión. Ahora bien, una tarea t_i es la última en ejecutarse si la misma no es origen de ninguna relación temporal *antes-que*. Nuevamente, esta afirmación queda claro, dado que de no existir esta relación no hay ninguna tarea t_j que se ejecute después que t_i. Dentro de una $TaskVersionDevs_V$ una tarea componente d_i es la última si no existe un acoplamiento entre el puerto de salida STA_{di} y el puerto de entrada ETA_{dj} para algún componente d_j perteneciente a $TaskVersionDevs_V$. No necesariamente debe existir una única tarea en una versión que sea la última, por lo tanto se define el conjunto *finalTask* como el conjunto de todas las tareas que se ejecutan últimas en una versión. Luego este conjunto se define como:

$$finalTask \quad = \quad \{d/d \,\epsilon\, D_V \;\wedge\; \nexists\, d_j \,\epsilon\, D_V/((d, STA_d)(d_j, ETA_{dj})) \;\in\; IC\} \tag{5.27}$$

Luego, los puertos de salidas STS_{di} y STA_{di} de todos los componentes d_i que pertenezcan al conjunto *finalTask* estarán acoplados a los puertos STS_V y STA_V de $TaskVersionDevs_V$. Esto se representa como sigue:

$$\begin{aligned} EOC_V \quad \supset \quad & \{((d, STS_d), (V, STS_V))\} \cup \{((d, STA_d), (V, STA_V))\} \\ & \forall d \,\epsilon\, finalTask \end{aligned} \tag{5.28}$$

Finalmente, los eventos que producen las tareas con valores estadísticos, deben ser propagados hacia arriba en la jerarquía de manera de alcanzar el modelo *ExperimentalFrame* encargado de recopilar y producir las estadísticas finales de una corrida de simulación. De aquí se desprende que los puertos TR_d de cada componente d debe estar acoplado al puerto TR_V de $TaskVersionDevs_V$. Esto queda definido por la siguiente expresión:

$$EOC_V \quad \supset \quad \{((d, TR_d)(V, TR_V))\} \,\forall\, d \,\epsilon\, D_V \tag{5.29}$$

5.4. SimulationModel

Este modelo representa el modelo de simulación que es generado como resultado del proceso de transformación del modelo conceptual. Como tal, debe representar el comportamiento del proceso que se seleccionó para analizar. Este modelo es un modelo acoplado y su comportamiento queda definido por las interacciones entre las tareas y recurso que participan del proceso. Sus componentes son los modelos de los recursos que participan de las diferentes versiones en los diferentes niveles de abstracción y la versión de tarea correspondiente al proceso seleccionado de mayor nivel. La figura 5.7 muestra un esquema genérico del modelo de simulación. En esta figura se pueden ver los componentes, sus puertos de entrada y salida y los acoplamientos. Por razones de simplicidad, se graficaron dos recursos llamados genéricamente R1 y Rn indicando que pueden existir n instancias de modelos *ResourceDevs* los cuales cumplirán con las propiedades de acoplamiento que estos modelos representan en el diagrama.

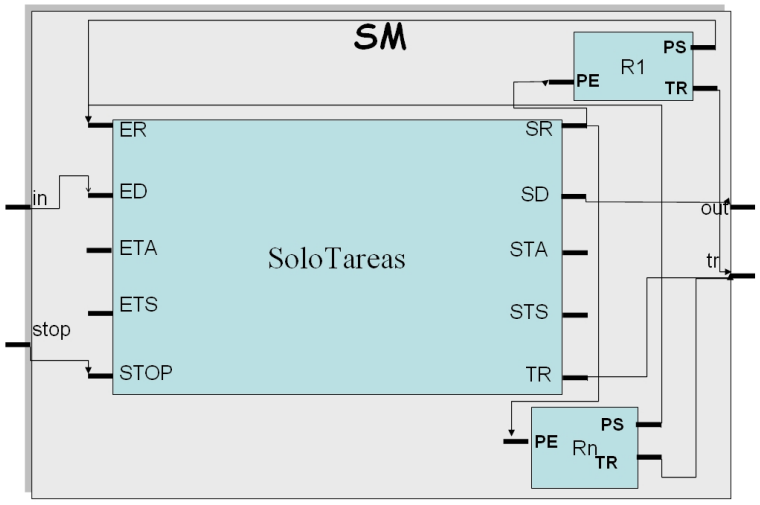


Figura 5.7: Esquema del modelo SimulationModel

Al ser el modelo de mayor nivel, tendrá puertos de entrada para recibir los eventos que forman el experimento que se ejecutará y puertos de salida por donde enviar los

resultados de la simulación. La ecuación 5.30 representa la tupla de los elementos que definen a este modelo.

$$SimulationModel =< X_{SM}; Y_{SM}; D_{SM}; \{M_i\}; EIC_{SM}; EOC_{SM}; IC_{SM} > \tag{5.30}$$

En los párrafos siguientes, se describen los elementos del modelo *SimulationModel*.

Conjunto de eventos de entrada ($\mathbf{X}_{SM}$). Este modelo tiene un puerto de entrada desde donde recibe los eventos que alimentan al sistema de simulación provenientes del experimento. A este puerto se lo llama *IN*. Otro puerto necesario es el puerto *STOP* por donde recibe el evento de fin de simulación. Luego, los puertos de entrada y los valores recibidos en estos puertos quedan definidos en el siguiente conjunto:

$$X_{SM} \quad = \quad \{(in_{SM}, process(ir)), (STOP_{SM}, stop)\} \tag{5.31}$$

El valor *process(ir)*, representa algún recurso que se genera externo al proceso. Por ejemplo puede identificarse una orden del cliente como un recurso que no es generado en el proceso sino que proviene de un cliente externo. Este recurso es el encargado de iniciar la simulación.

Conjunto de eventos de salida ($\mathbf{Y}_{SM}$). Este modelo tiene un puerto de salida por donde envía los resultados de la simulación, y otro puerto por donde envía eventos hacia otros simuladores cuando se trata de simulación distribuida. Los puertos de salida son llamados *tr* y *out* y los posibles valores en estos son:

$$\begin{aligned} Y_{SM} \quad = \quad & \{(tr_{SM}, waiting_r)(tr_{SM}, tSusp_t)(tr_{SM}, longCola_t), \\ & (tr_{SM}, notEnough_r), (out_{SM}, process(id))\} \end{aligned} \tag{5.32}$$

Los valores que aparecen en el puerto tr_{SM} son los valores que se utilizan para calcular las estadísticas, los mismos indican el tiempo que un recurso r estuvo en cola esperando a ser atendido por las diferentes tareas, el tiempo en que una tarea t estuvo suspendida, la longitud de cola promedio para una tarea t y la cantidad de tiempo que un recurso

r estuvo faltante. El puerto out_{SM} es el puerto de salida de eventos que serán enviados a otros simuladores en una simulación distribuida.

Conjunto de componentes ($\mathbf{D}_{SM}$). El *SimulationModel* tiene un conjunto de componentes entre los cuales pueden identificarse dos grupos: *SoloTareas* y el conjunto R de modelos de recursos. El modelo *SoloTareas* es un modelo *TaskVersionDevs* y está asociado a la tarea proceso de mayor nivel. El conjunto R de recursos comprende los modelos $ResourceDevs_i$ asociados a cada uno de los recursos que participan en el proceso. Luego, este conjunto de componentes queda representado por la siguiente expresión:

$$D_{SM} = SoloTareas \cup \{r_i / r_i\ is - a\ ResourceDevs\} \tag{5.33}$$

Acoplamiento externo de entrada ($\mathbf{EIC}_{SM}$). Los eventos que se reciben por el puerto de entrada *IN* deben ser analizados por las tareas que pertenecen a *SoloTareas*. Los eventos de entrada representan recursos que necesitan ser procesados de alguna manera por las tareas. Los eventos recibidos por el puerto *STOP*, indican el fin de la simulación y deben ser propagados a todos sus componentes. Luego:

$$\begin{aligned} EIC_{SM} = {} & ((SM, IN_{SM})(SoloTareas, ED_{SoloTareas})) \cup \\ & ((SM, STOP_{SM}), (SoloTareas, STOP_{SoloTareas})) \cup \\ & \{((SM, STOP_{SM})(r_i, ED_{ri}))\}\ \forall\ r_i \epsilon D_{SM} \end{aligned} \tag{5.34}$$

Así se establece que el puerto *IN* del modelo de simulación está acoplado con el puerto *ED* de *SoloTareas*, de esta forma todos los eventos que provengan del experimento serán recibidos por las tareas correspondientes. Por otro lado, el puerto *STOP* del modelo de simulación está conectado con el puerto *STOP* de todos sus componentes, esto es, el puerto *STOP* de *SoloTareas* y los puertos *STOP* de cada uno de los recursos que pertenecen al conjunto R.

Acoplamiento externo de salida ($\mathbf{EOC}_{SM}$). Tanto los recursos como también las tareas que participan de la simulación, generan eventos con valores estadísticos necesarios para poder generar los resultados de una corrida de simulación. Estos valores son propagados en el modelo a través de eventos especiales cuyo objetivo final es alcanzar el modelo encargado de hacer los cálculos estadísticos, este modelo es llamado *Resume* que será desarrollado en la sección 5.5.1.

Todos los modelos tienen un puerto especial llamado *TR* por donde envían estos eventos especiales. Así, es necesario que los puertos *TR* de cada componente del modelo *SimulationModel* esté acoplado con el puerto *TR* de éste. Por otro lado, los eventos generados por el modelo *SoloTareas* a través de su puerto *SD* deben ser propagados hacia otros simuladores a través del puerto de salida out_{SM}.

Luego, la siguiente expresión representa el acoplamiento externo de salida para este modelo:

$$\begin{aligned} EOC_{SM} &= ((SoloTareas, TR_{SoloTareas})(SM, tr_{SM})) \cup ((SoloTareas, SD_{SoloTareas})(SM, out_{SM})) \\ &\quad \cup\{((r_i, TR_{ri})(SM, tr_{SM}))\}\forall r_i \epsilon R \end{aligned} \tag{5.35}$$

El puerto de salida *TR* del modelo *SoloTareas*, tiene que estar conectada al puerto de salida del *SimulationModel* por donde enviarán eventos con datos necesarios para generar las estadísticas. De igual manera, los puertos *TR* de todos los recursos envían sus estadísticas a través del puerto de salida del modelo de simulación del que forman parte.

Acoplamiento interno ($\mathbf{IC}_{SM}$). El acoplamiento interno relaciona los elementos que pertenecen al modelo de simulación. Principalmente deben acoplarse las salidas de los modelos de los recursos con el modelo de la versión de mayor nivel (referenciada como *SoloTareas*), de manera que los eventos que envíen los recursos se propaguen hacia los componentes en la jerarquía de descomposición de tareas hasta alcanzar la tarea que lo usa. De igual manera, cuando una tarea envía eventos a sus recursos,

estos deben propagarse hacia los contenedores en la jerarquía hasta alcanzar al recurso correspondiente. Luego el acoplamiento interno queda definido como:

$$\begin{aligned} IC_{SM} &= \{((r_i, PS_{ri})(SoloTareas, ER_{SoloTareas}))\} \cup \\ &\quad \{((SoloTareas, SR_{SoloTareas}), (r_i, PE_{ri}))\} \forall r_i \epsilon R \end{aligned} \tag{5.36}$$

Esto es, los puertos de salidas *PS* de cada uno de los recursos estarán conectados al puerto de entrada *ER* de la *TaskVersionDevs SoloTareas* y el puerto de salida $SR_{SoloTareas}$ de la versión *SoloTareas* estará acoplado con cada uno de los puertos PE_{ri} de los recursos componentes del modelo de simulación.

5.5. ExperimentalFrame

Este modelo, conocido como marco experimental, representa el experimento con el que se prueba el modelo de simulación. Este modelo está definido en (Zeigler y otros, 2000b) y puede adoptar diferentes composiciones de acuerdo al experimento que se quiera lograr. En esta arquitectura el *ExperimentalFrame* es un modelo que tiene como finalidad determinar cuándo finalizar la simulación, alimentar al modelo de simulación con eventos adecuados y obtener métricas de interés. El modelo *ExperimentalFrame* (EF) se define como un modelo DEVS acoplado, formado por tres componentes: *Generator*, *Acceptor* y *Resume*. El gráfico de la figura 5.8 muestra un esquema de este modelo donde aparecen sus componentes, los acoplamientos entre ellos y los eventos de entrada/salida que se propagan por este sistema. El *Generator* es el encargado de generar los eventos del experimento. Para el caso del modelo que se está analizando, los eventos que se producen son recursos que deben alimentar el sistema. Por ejemplo, este modelo representa los eventos externos que causan la activación de los proceso de la empresa: (i) la llegada de una orden del cliente,(ii) ingreso de mercadería en depósito, (iii) la llegada de un pedido de reposición de mercadería desde un distribuidor, (iv) la

recepción de un pago. Estos pueden ser eventos que disparan una secuencia de tareas que se ejecutan. El modelo generador (llamado *Generator*) es el encargado de producir

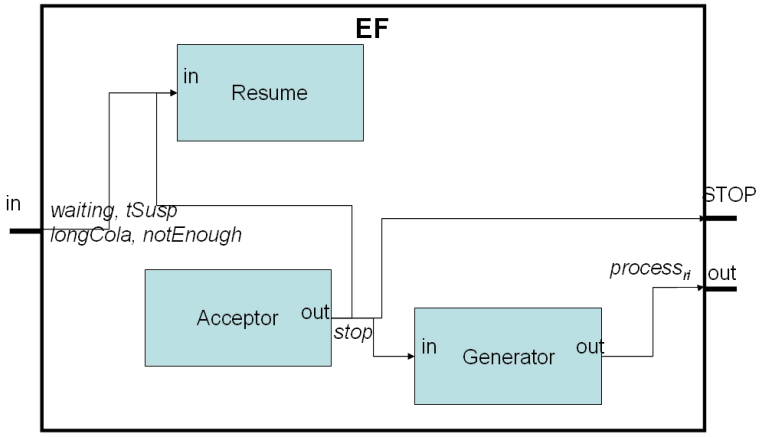


Figura 5.8: Modelo ExperimentalFrame

estos eventos, hasta que recibe el evento *stop* por su puerto de entrada proveniente del aceptor (modelo *Acceptor*). El aceptor, es el encargado de determinar cuándo la simulación termina. Este componente, contiene la condición de fin de simulación, que puede ser un tiempo específico o bien un estado determinado, y se encarga de monitorear el sistema de simulación en busca de la condición de finalización. Cuando esta condición es encontrada el mismo envía el evento *stop* para finalizar la simulación. El puerto de salida de *Acceptor* está acoplado con el puerto de salida *STOP* del modelo *ExperimentalFrame* de manera tal que el evento de finalización de la simulación sea propagado a todos los componentes del modelo de simulación. De esta manera, todos los modelos se informan que deben suspender sus actividades. Por último, el modelo *Resume* es el encargado de obtener las métricas y presentarlas de una manera adecuada. Cuando este modelo recibe el evento *stop* por su puerto de entrada calcula las estadísticas a partir de los datos que obtuvo durante la simulación. Este modelo, recibe constantemente los eventos que tareas y recursos envían por los puertos *TR* con las estadísticas. El

puerto de entrada *in* de *Resume* está acoplado con el puerto de entrada *in* del modelo *ExperimentalFrame*, para recibir los eventos desde el modelo de simulación.

El *ExperimentalFrame* tendrá un puerto de entrada *in* por donde recibe los valores de las estadísticas calculadas por los diferentes componentes (tareas y recursos) cuyo destino es el componente *Resume* encargado de obtener a partir de estos datos, estadísticas útiles para analizar el proceso. Los puertos de salidas son: (i) *stop* por donde se envía el fin de la simulación, (ii) *out*, por donde envía los eventos que alimentan al modelo de simulación, los cuales son los eventos generados por *Generator*.

Para poder cumplir con sus objetivos el modelo acoplado *ExperimentalFrame* cuenta con tres componentes cada uno con un comportamiento bien definido que se analiza a continuación.

5.5.1. Modelo Resume

Resume es un modelo DEVS atómico. Tiene un comportamiento especial: almacena los eventos que recibe de las distintas tareas y recursos con valores necesarios para calcular estadísticas. Sus transiciones son causadas por eventos externos únicamente. Así la llegada de un evento externo indica la necesidad de guardar el valor asociado al evento para futuros cálculos. El modelo *Resume* calcula para cada tarea, su longitud de cola y el tiempo promedio en que estuvo suspendida. Para cada recurso, calcula el tiempo en cola y el tiempo en que estuvo faltante.

$$Resume =< X_{res}, Y_{res}, \delta_{int_{res}}, S_{res}, \delta_{ext_{res}}, \lambda_{res}, ta_{res} > \tag{5.37}$$

Como puede observarse en la figura 5.8 *Resume* tiene un solo puerto de entrada. Así su conjunto X_R de puertos de entrada y sus valores posible se define como:

$$X_{res} = \{(in, waiting)(in, tSusp)(in, longCola)(in, stop)\} \tag{5.38}$$

Este modelo no tiene puertos de salida dado que no envía eventos a ningún componente, su función es almacenar valores y calcular valores de simulación. Por lo dicho se define

el conjunto Y_R como un conjunto vacío como sigue:

$$Y_{res} = \emptyset \tag{5.39}$$

Se establecieron dos estados posibles para este modelo: (i)*Recording* identifica el estado en que el modelo *Resume* se encuentra recibiendo eventos externos y almacenando estos valores en su estado interno; (ii) *final* el modelo lo alcanzará una vez que recibe el evento de fin de simulación.

Este modelo no tiene función de transición de estados interno y su función de transición de estado externa queda definida con el gráfico de la figura 5.9. Si el modelo se encuentra en el estado *recording*, y recibe alguno de los eventos externos *tSusp*, *longCola*, *notEnough* o *waiting*, hace una transición al mismo estado *recording* guardando los valores asociados a dichos eventos en su estado interno. Cuando recibe el evento de fin de simulación *stop*, pasa al estado *final*.

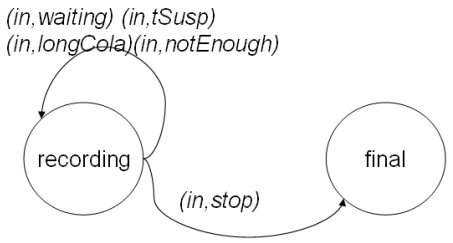


Figura 5.9: Función de transición externa del modelo Resume

La función de transición del tiempo (ta_{res} queda definida como:

ta_{res}(recording)= ∞

ta_{res}(final)= ∞

Esto significa que ambos estados son estados pasivos, pudiendo abandonarlos únicamente con la recepción de eventos externos.

Habiendo analizado el comportamiento de este modelo, se define a continuación las fórmulas de los valores estadísticos que este modelo genera. Con los valores guardados en su estado interno, proveniente de las tareas y recursos, el modelo *resume* calcula las estadísticas como sigue:

Para cada tarea i:

Tiempo promedio de inactividad TI_i y longitud de cola promedio LC_i:

$$TI_i = \frac{\sum_{j=0}^{n} tSusp_{ij}}{tiempoTotalSimulacion} \tag{5.40}$$

$$LC_i = \frac{\sum_{i=0}^{n} longCola_i}{n} \tag{5.41}$$

Tiempo total de inactividad del proceso: TIP

$$TIP = \sum_{i=0}^{n} TI_i \tag{5.42}$$

Para cada Recurso i:

Tiempo promedio en cola (TC_i):

$$TC_i = \frac{\sum_{i=0}^{n} waiting_i}{tiempoTotalSimulacion} \tag{5.43}$$

Tiempo promedio faltante (TF_i)

$$TF_i = \frac{\sum_{i=0}^{n} notEnough_i}{tiempoTotalSimulacion} \tag{5.44}$$

Luego, estos valores son graficados a través del componente *View* de la arquitectura.

5.5.2. Modelo Generator

Este modelo es el encargado de generar los eventos que alimentan al modelo de simulación. Estos eventos representan recursos que son generados externos al proceso que

se simula. En la construcción de este modelo, se utiliza información sobre los recursos que tiene una relación *procesa* con una tarea en el modelo conceptual, estos recursos se toman como patrón de generación de eventos *process(id)* donde *id* es una instancia de un modelo ResourceDevs correspondiente a uno de los recursos patrón que deben ser procesado por las tareas. Este evento se inyecta en el modelo de simulación a través del puerto *in*.

$$Generator =< X_G, Y_G, \delta_{int_G}, S_G, \delta_{ext_G}, \lambda_G, ta_G > \tag{5.45}$$

Principalmente *Generator* se identifica por sus transiciones internas, dado que su principal objetivo es producir eventos. Existe una única transición externa causada por la recepción del evento *stop* indicando fin de la simulación. Así pueden definirse los conjuntos X_G e Y_G como sigue:

$$X_G = \{(in, stop)\} \tag{5.46}$$

$$Y_G = \{(out, process(id))\} \tag{5.47}$$

Para este caso, *id* es un modelo *ResourceDevs* correspondiente al recurso patrón que se identifica.

Para comprender la manera en que este modelo genera los eventos necesarios para alimentar el modelo de simulación, la figura 5.10 muestra el diagrama de transición de estado de este modelo. Como se observa existen dos estados: *generating* y *final*.

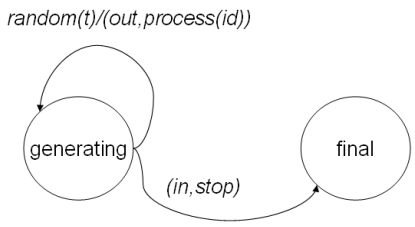


Figura 5.10: Diagrama de transición de estados del modelo Generator

Un dato importante en la construcción de este modelo es el tiempo asignado al estado *generating*. Cada vez que se produce una transición interna, esto es, se cumple

el tiempo en el estado *generating*, el modelo genera el evento de salida y realiza la transición interna, que para el caso particular de este modelo, seguirá estando en el mismo estado *generating* hasta que recibe un evento externo que lo haga evolucionar al estado *final*. Cada evento de salida de este modelo representa un evento de entrada al modelo de simulación, por lo tanto, la generación de estos eventos debe darse en forma aleatoria siguiendo alguna distribución de probabilidad. Para el caso del modelo *Generator* se utiliza una distribución de probabilidad uniforme donde el intervalo en el cual el evento será generado es un parámetro a determinar en el momento de ejecutarse la simulación. De igual manera, la distribución de probabilidad podría ser modificada sin alterar por ello el comportamiento del modelo que se define.

Así, en cada transición interna, la función del tiempo *ta(generating)* es calculada de acuerdo al nuevo valor generado por la función *random*.

La descripción realizada se expresa mediante el siguiente modelo DEVS:

Función de transición de estados interna se define como:

$$\delta_{int_G}(generating) = generating \tag{5.48}$$

indica que el estado sucesor de *generating* es el mismo estado.

La función de transición de estados externa se define como:

$$\delta_{ext_G}(generating, e, (in, stop)) = final \tag{5.49}$$

Indica que estando en el estado *generating*, al transcurrir e instantes de tiempo después de la última transición, y habiendo recibido el evento *stop* en el puerto *in* el nuevo estado es *final*.

La función de avance del tiempo:

$$ta_G(generating) = uniform(p) \tag{5.50}$$

indica que el tiempo en que el modelo está en el estado *generating*, es un valor aleatorio que se calcula de acuerdo a una distribución de probabilidad, en este caso uniforme,

siendo p el parámetro que varía de acuerdo al caso específico que se quiera tratar. Así por ejemplo si se quiere generar eventos cada 20´ entonces, p tomará el valor 20. Luego el valor uniform(p) será un valor en el intervalo [18-22] indicando que el próximo evento será generado transcurrido dicho tiempo.

Finalmente se define la función de salida como sigue:

$$\lambda_G(generating) = (out, process(id)) \tag{5.51}$$

Esta función indica que el estado *generating* produce un evento *process(id)* por el puerto *out* antes de proceder a realizar la transición interna.

5.5.3. Modelo Acceptor

Este modelo es un modelo DEVS atómico, que al inicializarse, su estado es *suspended* y permanece en el mismo por un tiempo igual al tiempo de simulación. Cuando este tiempo expira, el modelo genera el evento externo *stop* que se propaga por todo el modelo de simulación provocando que sus componentes suspendan sus actividades. Luego, pasa a un estado *final* por un tiempo infinito. De este modo, este modelo es el que determina cuándo detener la simulación.

$$Acceptor =< X_A, Y_A, \delta_{int_A}, S_A, \delta_{ext_A}, \lambda_A, ta_A > \tag{5.52}$$

El gráfico de la figura 5.11 muestra el diagrama de transición de estados que resume su comportamiento.

De esta forma cuando se determina el tiempo total de simulación, éste será en el cual el modelo *Acceptor* estará en el estado *suspended* sin realizar ninguna actividad. Luego cuando este tiempo se cumple, *Acceptor* realiza la transición de estado interna

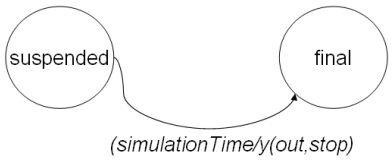


Figura 5.11: Diagrama de transición de estado del modelo Acceptor

generando el evento *stop* por el puerto de salida *out* y queda en el estado *final*.

El modelo *Acceptor* no tiene puertos de entrada dado que no necesita recibir eventos de otros modelos, luego el conjunto X_A queda definido como:

$$X_A = \emptyset \tag{5.53}$$

Su puerto de salida es uno solo: *out* por donde envía el evento de fin de simulación. El conjunto Y_A se define como:

$$Y_A = \{(out, stop)\} \tag{5.54}$$

De acuerdo a la descripción previa, la función de transición interna queda definida como:

$$\delta_{int_A}(suspended) = final \tag{5.55}$$

indicando que el estado sucesor de *suspended* es *final* dado que transcurrió el tiempo en que el modelo debe estar en el estado suspended. No tiene transición de estado externa dado que no recibe eventos externos. Finalmente se representa su función de avance de tiempo como:

$\lambda_A(suspended) = tiempoSimulacion$

$\lambda_A(final) = \infty$

De esta manera, se ha establecido que el tiempo en el estado *suspended* es igual al tiempo de simulación, el cual es un parámetro que se especifica en el momento de

ejecutarse la simulación.

5.6. Conclusiones

Primeramente se presentaron las definiciones conceptuales de los bloques de construcción usados en el modelo de simulación. Los mismos fueron descriptos reflejando las relaciones existentes entre estos bloques y los elementos conceptuales que los mismos representan. Se presentaron y analizaron las interfaces de cada uno de los bloques de construcción representadas por el conjunto de puertos y los eventos que por los mismos se reciben. Cada bloque puede considerarse como una unidad independiente que puede ser probado y analizado aisladamente. Los diferentes bloques se presentan como modelos DEVS siguiendo la sintaxis propuesta por el formalismo. Se hizo una descripción detallada de cada uno de los bloques identificando las entradas, salidas, y comportamiento interno, es decir, cómo reacciona el modelo ante eventos internos y externos. Se dio a conocer la composición de los modelos acoplados explicándose el comportamiento de los mismos.

6

Modelo de simulación

En este capítulo se aborda la transformación del modelo conceptual en un modelo de simulación y se presenta el diseño de la capa de simulación (Simulation Layer) de la arquitectura.

El modelo de simulación (SM) es generado por los componentes de la capa de simulación (Simulation Layer), la cual presta servicios para generar y ejecutar el modelo de simulación a partir de un modelo conceptual concebido con el lenguaje Coordinates. La arquitectura fue diseñada de manera de dar soporte tanto a la ejecución local del SM como también a la ejecución distribuida. Para este último caso, se considera al SM como un componente que puede ser acoplado a otros SM formando un modelo de nivel superior, el cual puede ser ejecutado en un entorno distribuido bajo el estándar HLA.

En la construcción del modelo de simulación, se utiliza el formalismo DEVS, dado que el mismo permite una construcción modular y jerárquica de los modelos. Pero fundamentalmente, la característica más importante de DEVS para ser usado en esta arquitectura es sin dudas, la posibilidad de separar los modelos de la máquina de simulación que los interpreta. La primera de estas características tiene relación con la estructura que presentan los EM desarrollados usando Coordinates. Principalmente, el proceso de construcción del SM se centra en los modelos de Tareas que conforman el

EM. Estos modelos, son modulares y jerárquicos en el sentido que las tareas se descomponen en tareas más simples. Consecuentemente es posible imitar en la construcción del modelo de simulación la estructura del modelo conceptual de empresa evitando ambigüedades y errores.

La segunda característica está relacionada al concepto de reuso de componentes. En este sentido, el SM es generado una sola vez, y reusado en diferentes entornos: local y distribuido. Para ello, el mismo SM es interpretado por máquinas de simulación diferentes conduciendo al reuso del SM.

Ahora bien, estas características se complementan, dando como resultado la posibilidad de representar el EM, no como una unidad en sí misma, sino como un módulo o componente que puede ser integrado a otros EM, generando así, un modelo de nivel superior, que representa por ejemplo una empresa virtual, empresa en red o empresa distribuida. La forma de integrar estos modelos es a través de su comportamiento reflejado en el modelo de simulación. De esta forma, es posible interpretar una simulación distribuida como el comportamiento de una empresa que está formada por un conjunto de componentes (otras empresas o ramas de la organización), que interactúan, pero que a su vez tienen independencia y comportamiento autónomo. De estas características surge el concepto dado en el capítulo 1 de DE^2M representando un modelo de empresa, ejecutable y distribuido, donde los modelos conceptuales de la empresa (EM) son ejecutados a través de máquinas de simulación que pueden interactuar en forma distribuida generando el comportamiento de una empresa que comprende la forma de proceder de cada uno de los miembros que participa en la simulación distribuida.

Así, por ejemplo, el modelado de una cadena de suministro puede ser tratado como la definición de EM en cada uno de sus miembros, los cuales serán integrados a través de sus SM correspondientes, dando como resultado el comportamiento de la cadena de suministro.

Esta perspectiva facilita la construcción de los modelos dado que preserva la integridad y la autonomía de sus miembros, a la vez que permite obtener una visión global de la misma.

El SM generado con la arquitectura tiene la particularidad de poder interactuar con otros SM sin restricción del lenguaje con que estos otros fueron generados. Esto es, no es condición necesaria que los SM que interactúan estén generados usando el formalismo DEVS como los generados por la arquitectura propuesta.

Esta característica contribuye a la interoperabilidad de los simuladores y consecuentemente facilita la construcción de los DE^2M, dado que no es necesario que todos los integrantes de una simulación distribuida usen la misma herramienta para generar sus modelos de simulación. La única condición necesaria de interoperabilidad es que los simuladores puedan interpretar correctamente las interacciones y los objetos intercambiados entre ellos, tanto sintáctica como semánticamente.

HLA provee interoperabilidad sintáctica, los roles usados en esta arquitectura proveen de interoperabilidad semántica. Más adelante se tratará este último punto con más detalles.

La organización de este capítulo, se realiza de la siguiente manera. Primero, se presentan las reglas de transformación usadas para obtener el SM a partir del EM. A continuación se describe la arquitectura de la capa de simulación (Simulation layer) para dar soporte a la construcción y ejecución del SM. Finalmente, se detalla la forma de llevar a cabo la simulación distribuida, indicando las modificaciones realizadas en la máquina de simulación DEVS.

6.1. Transformación del Modelo Conceptual de Empresa en un Modelo de Simulación

El proceso de transformación está basado en el esquema mostrado en la figura 6.1. Así un modelos de empresa Coordinates es transformado en un modelo de simulación DEVS, a través de una herramienta de transformación que utiliza reglas expresadas en OCL. Tanto el lenguaje Coordinates como DEVS (es decir lenguajes fuentes y destino), están representados a través del lenguaje UML. Este proceso se lo realiza utilizando una estrategia MDA (Model Drive Architecture (Siegel, 2001)).

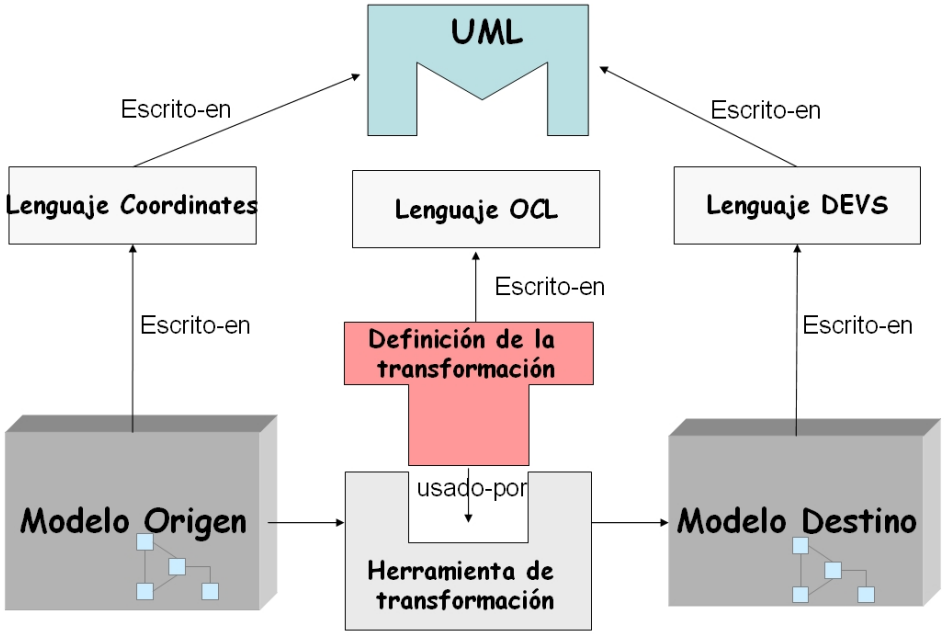


Figura 6.1: Esquema de transformación de EM en SM

La traducción del modelo de empresa se centra en los modelos de tareas, los cuales representan la descomposición de tareas en subtareas a través del concepto de Versión de tarea. Por lo tanto para comenzar con la transformación, se parte de la versión de tareas de mayor nivel que involucra el conjunto de los procesos definidos en la

organización. Luego a partir de esta versión, los diagramas de recursos y los ciclos de vida de los mismos son utilizados para representar los modelos de simulación de dichos componentes. Es importante destacar que el proceso de transformación considera la elaboración de un modelo de simulación capaz de ejecutarse en un entorno local, luego en caso de solicitar el servicio de simulación distribuida, el modelo es reusado para formar parte de un federado que tiene características particulares. Para dirigir la traducción de un modelo de empresa en un modelo de simulación se definieron reglas de transformación expresadas en el lenguaje formal OCL para MDA (Warmer y Kleppe, 2003). Estas reglas establecen un mapeo (o relación) entre elementos del lenguaje origen y elementos del lenguaje destino, basados en la definición de dichos lenguajes. Los lenguajes que se utilizan están expresados en UML, el cual representa un metamodelo de los lenguajes referenciados en la transformación. En este contexto OCL juega un rol importante dado que permite definir los modelos con mayor precisión, representar definiciones de los lenguajes de modelado y finalmente definir las reglas de transformación.

El diagrama de la figura 6.2 muestra las relaciones entre los elementos del lenguaje fuente Coordinates y los del destino DEVS utilizando UML.

En este diagrama de clases se puede observar por ejemplo, que la clase *TaskVersion*, correspondiente al concepto de versión de tarea en Coordinates, está relacionada con la clase *TaskVersionDevs*, correspondiente al modelo Devs acoplado que representa a dicha versión en el modelo de simulación. La relación que vincula a ambas clases tiene en sus extremos el nombre de los roles de dicha relación: +model y +devs. Esto indica que la clase *TaskVersion* tiene un modelo devs asociado (rol +devs) el cual es una instancia de *TaskVersionDevs*. A su vez, como la relación es bidireccional, la clase *TaskVersionDevs* tiene un modelo asociado (rol +model) que corresponde a la entidad conceptual en el modelo Coordinate que ésta representa. Luego, para poder hacer referencia al modelo de simulación asociado a una versión de tarea en Coordinates es posible identificarlo a

través de la siguiente expresión: *TaskVersion.devs* en donde se utiliza el nombre de la clase seguida del nombre del rol en la relación que se estableció entre ambas clases. En otras palabras, es posible a partir de la clase *TaskVersion* seguir la línea que representa la relación hasta encontrar el rol *+devs* en el otro extremo, la línea termina en la clase *TaskVersionDevs*, así se interpreta que el resultado de *TaskVersion.devs* es una instancia de la clase *TaskVersionDevs*.

A continuación se presentan las siete reglas que especifican el proceso de transformación. Para presentar cada regla, se hace una breve descripción de la misma y se la especifica empleando el lenguaje OCL para MDA.

Regla 1. *A partir de la versión de tarea que fue seleccionada para simular, se crea un digrafo f correspondiente al modelo de mayor nivel, el mismo tiene dos componentes: el ExperimentalFrame (ef) y el SimulationModel (sm)*

Gráficamente esta regla puede expresarse como se muestra en la figura 6.3, donde a partir de una versión de tarea *v1*, la misma es transformada dando origen al modelo de mayor nivel llamado *f*, el cual es un digrafo (instancia de la clase *digraph*) que tiene dos componentes: el modelo de simulación (sm instancia de la clase *SimulationModel*) y el experimento (ef instancia de la clase *ExperimentalFrame*). Notar que *f* es necesario dado que de no incluir *sm* y *ef* dentro de un modelo acoplado no se podrían establecer los acoplamientos entre los mismos.

Esta regla queda expresada formalmente como sigue:

Transformation TaskVersionToFramework(Coordinates, DEVS)
Source:

$$v1 : Coordinates :: TaskVersion \tag{6.1}$$

Target

$$\begin{aligned} f &: DEVS :: digraph \\ sm &: DEVS :: SimulationModel \\ ef &: DEVS :: ExperimentalFrame \end{aligned} \tag{6.2}$$

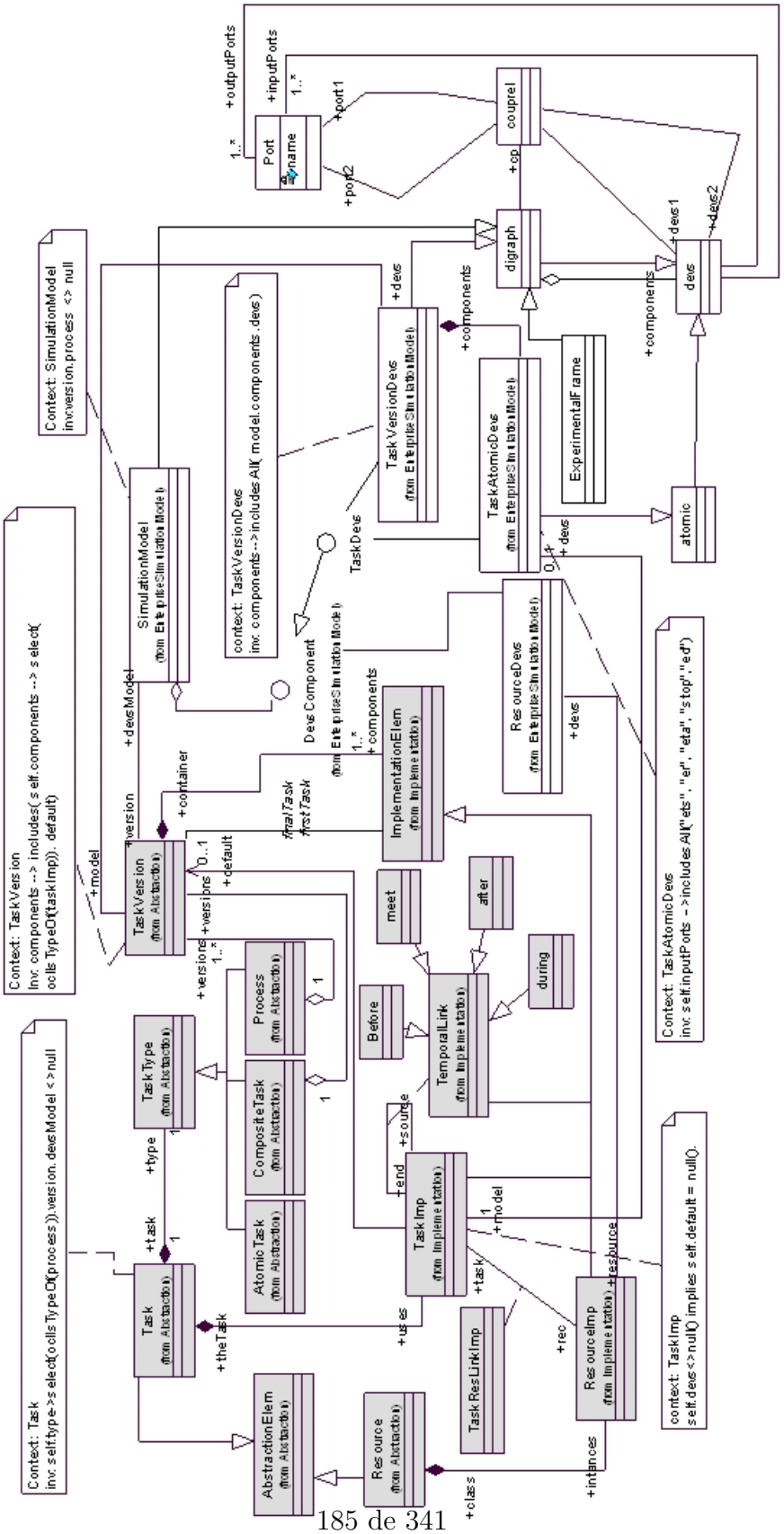


Figura 6.2: diagrama de clases Devs-Coordinates

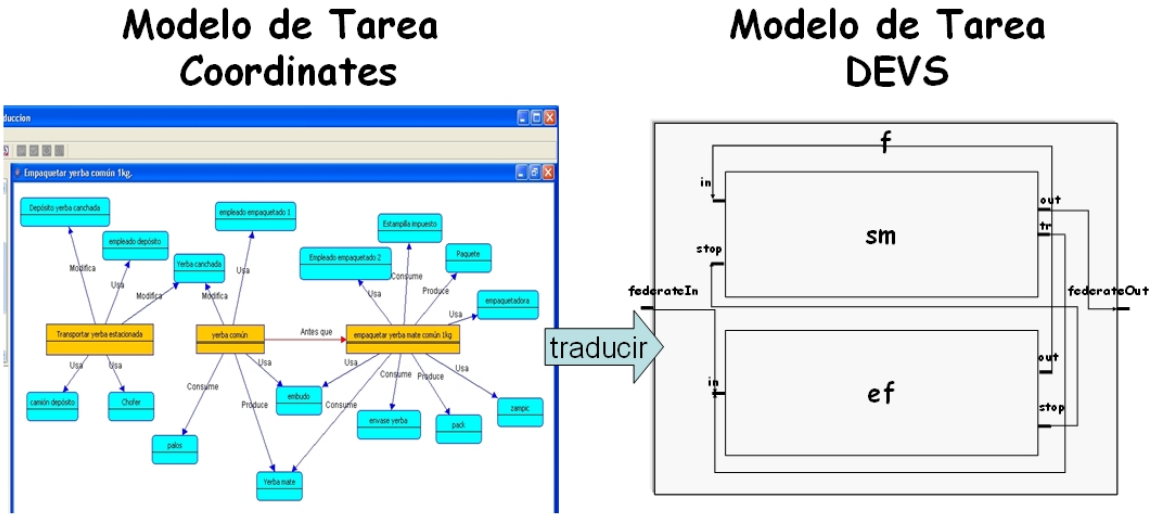


Figura 6.3: Transformación de Versión de tarea en un digrafo

Source condition
Target condition

$$sm.version = v1 \quad (6.3)$$
$$f.components \rightarrow includes(ef, sm) \quad (6.4)$$
$$f.inputports.name \rightarrow includes(\text{“}federateIn\text{”}) \quad (6.5)$$
$$f.outports.name \rightarrow includes(\text{“}federateOut\text{”}) \quad (6.6)$$
$$sm.inputports.name \rightarrow includes(\text{“}in\text{”}, \text{“}stop\text{”}) \quad (6.7)$$
$$sm.outputport.name \rightarrow includes(\text{“}tr\text{”}) \quad (6.8)$$
$$ef.inputports.name \rightarrow includes(\text{“}in\text{”}) \quad (6.9)$$
$$ef.outputports.name \rightarrow includes(\text{“}out\text{”}, \text{“}stop\text{”}) \quad (6.10)$$

- - *acoplamiento interno entre sm y ef:*

$$f.cp \rightarrow exists\,(t\,|t.desv1 = ef \wedge t.port1.name = \text{“}out\text{”} \wedge t.devs2 = sm \wedge t.port2.name = \text{“}in\text{”}) \quad (6.11)$$

$$f.cp \rightarrow exists\,(t\,|t.desv1 = ef \wedge t.port1.name = \text{“}stop\text{”} \wedge t.devs2 = sm \wedge t.port2.name = \text{“}stop\text{”}) \quad (6.12)$$

$$f.cp \rightarrow exists(t\,|t.desv1 = sm \wedge t.port1.name = \text{“}tr\text{”} \wedge t.devs2 = ef \wedge t.port2.name = \text{“}in\text{”}) \quad (6.13)$$

- - *acoplamiento externo de entrada*

$$f.cp \rightarrow exists(t\,|t.desv1 = f \wedge t.port1.name = \text{“}federateIn\text{”} \wedge t.devs2 = ef \wedge t.port2.name = \text{“}in\text{”}) \quad (6.14)$$

- - acoplamiento externo de salida

$$f.cp \quad \rightarrow \quad exists(t\,|t.desv1 = sm \wedge t.port1.name = \text{“}out'' \wedge \\ t.devs2 = f \wedge t.port2 = \text{“}federateOut'') \tag{6.15}$$

Mapping
try TaskVersionToSimulationModel **on**

$$v1.components <\sim> f.sm \tag{6.16}$$

A continuación se describe brevemente la sintaxis con que se enuncian las reglas de transformación.

La regla comienza con la palabra clave **Transformation** seguida del nombre de la regla (*TaskVersionToFramework*) con dos argumentos que indican los lenguajes de origen y destino de los modelos a transformar. En este caso particular se transforma desde un modelo escrito en Coordinates a un modelo escrito en DEVS. Las palabras claves **Source** y **Target** indican los modelos fuentes y destino de la transformación. Así, esta regla transforma un modelo del tipo *TaskVersion* (expresión 6.1) y como resultado de la transformación surgen los modelos que aparecen definidos en *Target* (expresión 6.2), es decir un digrafo *f*, instancia de la clase *digraph*, un modelo de simulación *sm*, instancia de la clase *SimulationModel*, y un marco experimental *ef*, instancia de al clase *ExperimentalFrame*.

Se pueden observar estos modelos generados en el diagrama de la figura 6.2. Así *f* es una instancia de la clase *digraph*, dicha clase tiene como componentes instancias de la clase *devs* lo cual queda expresado a través de la relación de composición con rol *+components*. Luego, *sm* y *ef* son instancias de las clases *SimulationModel* y *ExperimentalFrame* respectivamente, ambas clases son subclases de *digraph* la cual es a su vez subclase de la clase *devs*.

Source condition y **Target condition** establecen las condiciones que deben ser encontradas en los modelos origen y destino de la transformación para que la misma tenga efecto. En esta regla en particular no se han definido condiciones para los modelos

fuentes. Esto es, dado una versión de tarea, esta puede ser transformada sin ningún tipo de restricción. En cambio para los modelos resultados de la transformación, se establecen restricciones que deben cumplirse una vez que los modelos obtenido fueron creados. Así, la primer condición establece que el modelo *sm*, tiene asociado la versión de tarea *v1* a través de su relación *version* (expresión 6.3). Luego, la expresión 6.4 establece que los componentes de *f* son los modelos *sm* y *ef*. Seguidamente las expresiones 6.5 a 6.10 determinan los puertos de entrada y salida para los modelos *f, sm* y *ef*. Las expresiones 6.11, 6.12, y 6.13 describen los acoplamientos internos entre *ef* y *sm*. Estos acoplamientos determinan de que manera los componentes de *f* tienen sus puertos de entrada y salida conectados entre sí. Así por ejemplo, la condición 6.11, establece que el digrafo **f** tiene entre el conjunto de relaciones de acoplamientos (cp) un elemento que identifica el acoplamiento entre el puerto de salida con nombre *"out"* del componente **ef** (ExperimentalFrame) con el puerto de entrada llamado *"in"* del componente **sm** (SimulationModel). De esta forma, los acoplamientos se expresan como pares de la forma: ((modelo origen, nombre puerto del modelo origen); (modelo destino, nombre puerto modelo destino)).

Para poder identificar en el diagrama de la figura 6.2 estas relaciones de acoplamiento, se necesita centrar la atención sobre la clase *digraph* de la cual *f* es una instancia. Se observa que esta clase tiene una relación con la clase *couprel* y el rol que se establece en ésta se identifica como *+cp*, esto quiere decir que *f* tiene un conjunto *cp* de relaciones de acoplamiento. Ahora bien, si se focaliza sobre la clase *couprel* se puede observar que la misma tiene dos relaciones con la clase *Port* y dos con la clase *devs*. Las relaciones con la clase *Port* son identificadas como *+port1* y *+port2*, mientras que las relaciones con la clase *devs* son identificada como *+devs1* y *+devs2*. A su vez la clase *Port* tiene un atributo llamado *name* que identifica el nombre del puerto. Estas relaciones forman los pares que definen la relación de acoplamiento identificadas co-

mo: *((+devs1,+port1.name); (+devs2, +port2.name))*, donde *+port1.name* identifica el nombre del puerto interviniente en la relación.

Las últimas expresiones dentro de las restricciones de los modelos destinos se encuentran las expresiones 6.14 y 6.15 que establecen los acoplamientos externos de entrada y externos de salida respectivamente, siguiendo la notación explicada en el párrafo anterior.

Finalmente, la palabra **Mapping** establece la forma de continuar la transformación indicando nuevas reglas de transformación. En esta primer regla se tiene solo una regla de mapeo (expresión 6.16) la cual puede leerse como: "*intente transformar los componentes de la versión v1 en el modelo de simulación sm, siguiendo las condiciones establecidas en la regla* ***TaskVersionToSimulationModel***". De esta manera, la regla **TaskVersionToSimulationModel** describe la transformación de los componentes de la versión de tarea *v1* en el modelo de simulación *sm*. Como se puede observar, la construcción del modelo *sm* es desarrollado top-down, manteniendo en el modelo resultado la estructura de descomposición que presentan los modelos fuentes. La regla *TaskVersionToSimulationModel* corresponde a la regla 2, que se presenta a continuación.

Regla 2: *Los componentes de la versión de tarea, se transforman en los componentes del modelo de simulación.*

Esta regla queda expresada con el gráfico de la figura 6.4 en el que se observa cómo los recursos son transformados en componentes del *sm* y el conjunto de tareas da origen al componente *SoloTareas*.

Transformation TaskVersionToSimulationModel (Coordinates, DEVS)
Source

$$v1 : Coordinates :: TaskVersion \tag{6.17}$$

Target

$$\begin{aligned} sm &: DEVS :: SimulationModel \\ SoloTareas &: DEVS :: TaskVersionDevs \end{aligned} \tag{6.18}$$

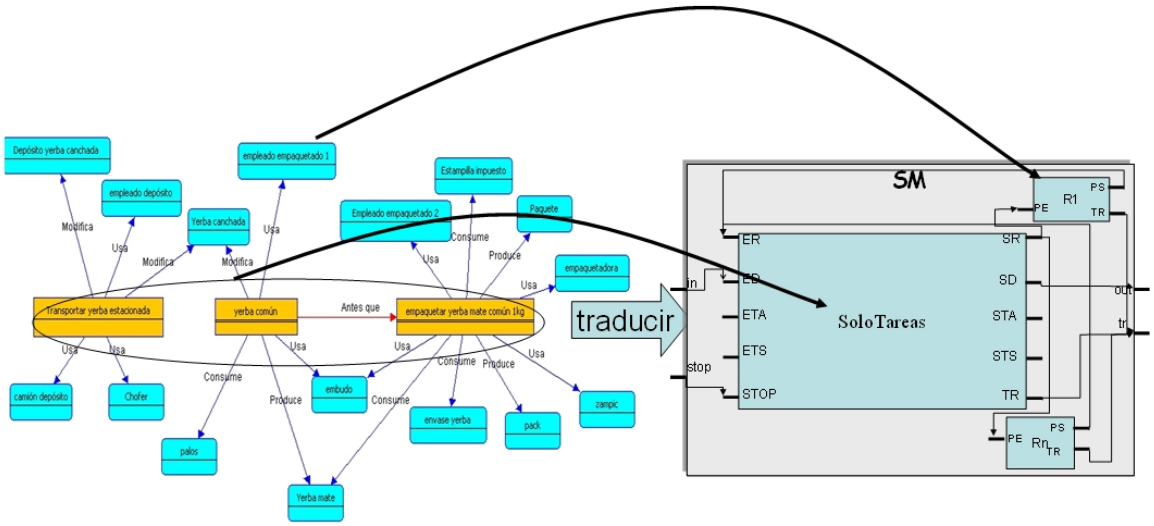


Figura 6.4: Diagrama de transformación de los componentes de la Versión de tareas en el modelo de simulación

Source condition
Target condition

$$SoloTareas.inputports.name \rightarrow includes(\text{“}er\text{”}, \text{“}eta\text{”}, \text{“}ets\text{”}, \text{“}ed\text{”}, \text{“}stop\text{”}) \quad (6.19)$$

$$SoloTareas.outputport.name \rightarrow includes(\text{“}sr\text{”}, \text{“}sts\text{”}, \text{“}sta\text{”}, \text{“}sd\text{”}, \text{“}tr\text{”}) \quad (6.20)$$

- - los componentes son: el modelo SoloTareas y los modelos DEVS de los recursos que participan en la versión:

$$sm.components \rightarrow includes(SoloTareas) \quad (6.21)$$

$$sm.components \rightarrow includes(v.components \rightarrow select(OclIsTypeof(ResourceImp)).devs() \quad (6.22)$$

- - acoplamientos externo de entrada entre sm y SoloTareas:

$$sm.cp \rightarrow exists(t\,|t.desv1 = sm \wedge t.port1.name = \text{“}in\text{”} \wedge t.devs2 = SoloTareas \wedge t.port2.name = \text{“}ed\text{”}) \quad (6.23)$$

$$sm.cp \rightarrow exists(t\,|t.desv1 = sm \wedge t.port1.name = \text{“}stop\text{”} \wedge t.devs2 = SoloTareas \wedge t.port2.name = \text{“}stop\text{”}) \quad (6.24)$$

- - acoplamiento externo de salida entre sm y SoloTareas:

$$sm.cp \rightarrow exists(t\,|t.desv1 = SoloTareas \wedge t.port1.name = \text{“}tr\text{”} \wedge t.devs2 = sm \wedge t.port2.name = \text{“}tr\text{”}) \quad (6.25)$$

$$sm.cp \rightarrow exists(t\,|t.desv1 = SoloTareas \wedge t.port1.name = \text{“}sd\text{”} \wedge t.devs2 = sm \wedge t.port2.name = \text{“}out\text{”}) \quad (6.26)$$

– *acoplamientos internos:*

$$sm.cp \rightarrow exists(c\,|c.desv1 = SoloTareas \wedge c.port1.name = \text{“}sr'' \wedge c.devs2 = r \wedge c.port2.name = \text{“}pe'' \wedge r.isOclType(ResourceDevs)) \quad (6.27)$$

$$sm.cp \rightarrow exists(c\,|c.desv1 = r \wedge c.port1.name = \text{“}ps'' \wedge c.devs2 = SoloTareas \wedge c.port2.name = \text{“}er'' \wedge r.isOclType(ResourceDevs)) \quad (6.28)$$

Mapping
try TaskToTaskDevs **on**

$$v1.taskImp <\sim> SoloTareas.components \quad (6.29)$$

try ResourceToResourceDevs **on**

$$v1.resourceImp <\sim> sm.resourceDevs \quad (6.30)$$

try VersionToVersionDevs **on**

$$v1.taskImp.default <\sim> SoloTareas.components \quad (6.31)$$

try TemporalLinkToCoupling

$$v1.taskImp.temporalLink <\sim> SoloTareas.cp \quad (6.32)$$

try TaskResLinkToCoupling **on**

$$v1.taskImp.taskResLinkImp <\sim> SoloTareas.cp \quad (6.33)$$

El modelo origen de esta regla, es la versión de tareas cuyos componentes serán transformados. El resultado final, es el modelo de simulación *sm* ahora definido con sus componentes. Nótese que en la primera regla, el modelo de simulación aparece como un componente del modelo de mayor nivel (sm en la figura 6.3) y de cómo éste se relaciona con el marco experimental para la simulación. Pero no se establece cómo el modelo de simulación está conformado, dado que esto corresponde a la Regla 2. Consecuentemente, los resultados de la Regla 2 son el modelo de simulación *sm* y el modelo *TaskVersion-Devs* llamado *SoloTareas*, el cual representa la descomposición jerárquica de las tareas dentro de la versión sin tener en cuenta las relaciones con los recursos. Las expresiones 6.19 y 6.20 describen cuales son los puertos de entrada y salida para *SoloTareas*. La expresión 6.21 establece que *SoloTareas* es un componente del *sm*. Otros componentes que forman parte de *sm* son los modelos de simulación correspondientes a los recursos

que participan en la versión *v1*. Así, la expresión 6.22 establece que dentro del conjunto de componentes del modelo de simulación (referenciado como *sm.components*), están incluidos los modelos *devs* asociados a las instancias de *ResourceImp* que son componentes de la versión *v1* analizada. Finalmente se identifican los acoplamientos externos de entrada y de salida como también los acoplamientos internos entre los modelos componentes y el modelo contenedor. Las restricciones 6.23 y 6.24 describen el acoplamiento externos de entrada entre *sm* y *SoloTareas*. La primera de estas, 6.23, establece que el puerto *in* de *sm* está acoplado con el puerto *ed* de *SoloTareas*. Así, los eventos recibidos por el puerto *in* provenientes del marco experimental, son enviados al modelo *SoloTareas* que representa la descomposición de las tareas para su tratamiento. La segunda restricción, 6.24, identifica el acoplamiento entre el puerto *stop* de *sm* y el puerto *stop* de *SoloTareas*, de manera tal que el evento de fin de simulación enviado desde el marco experimental al *sm* sea propagado hacia el conjunto de tareas. El acoplamiento externo de salida está definido por las expresiones 6.25 y 6.26 que establecen el acoplamiento entre el puerto *tr* de *SoloTareas* y el puerto *tr* de *sm* y el acoplamiento entre el puerto *sd* de *SoloTareas* y el puerto *out* de *sm* respectivamente. El primero de estos acoplamientos permite que los eventos generados por las tareas se propaguen hasta llegar al marco experimental donde estos eventos serán tratados. Finalmente se establecen los acoplamientos internos entre los componentes de *sm*. La expresión 6.27 establece que para cada componente de *sm* del tipo *ResourceDevs*, su puerto de salida *sr* esta conectado con el puerto de entrada *er* de *SoloTareas* y el puerto de salida *sr* de *SoloTareas* esta conectado con el puerto de entrada *er* de cada uno de los recursos participantes.

Dado que la versión *v1* puede tener como componentes tareas, recursos y relaciones, las reglas de mapeo en la sección *Mapping* establecen cómo cada componente de la versión será transformado en un componente del modelo de simulación de acuerdo a

otras reglas de transformación tales como:

TaskToTaskDevs indica cómo una tarea en el modelo conceptual es transformada en una *TaskDevs* que participa del modelo de simulación (6.29)

ResourceToResourceDevs indica cómo un recurso en el modelo conceptual es transformado en un *ResourceDevs* que participa del modelo de simulación (6.30)

VersionToVersionDevs indica cómo una versión de tarea asociada a través del vínculo *default* es transformada en una *TaskVersionDevs* que participa del modelo de simulación (6.31).

TemporalLinkToCoupling indica cómo las relaciones temporales existentes entre las tareas se transforman en acoplamientos entre los modelos DEVS que forman el modelo de simulación (6.32).

TaskResLinkToCoupling indica cómo las relaciones entre tareas y recursos se transforman en acoplamientos entre los modelos DEVS que forman el modelo de simulación (6.33).

La Regla 3 que se explica a continuación corresponde a la transformación de cada una de las tareas participantes del modelo conceptual.

Regla 3: *Cada implementación de tarea que participa de una versión de tarea, se transforma en un modelo DEVS, AtomicTaskDevs, cuyo tiempo de ejecución es igual a la duración de la tarea en el modelo conceptual.*

El diagrama de la figura 6.5 muestra la transformación de una tarea atómica (denominada *sapecado*), que participa de una versión de tareas, en una *AtomicTaskDevs* que forma parte del modelo *SoloTareas*.

Esta regla se especifica formalmente con el siguiente conjunto de expresiones:

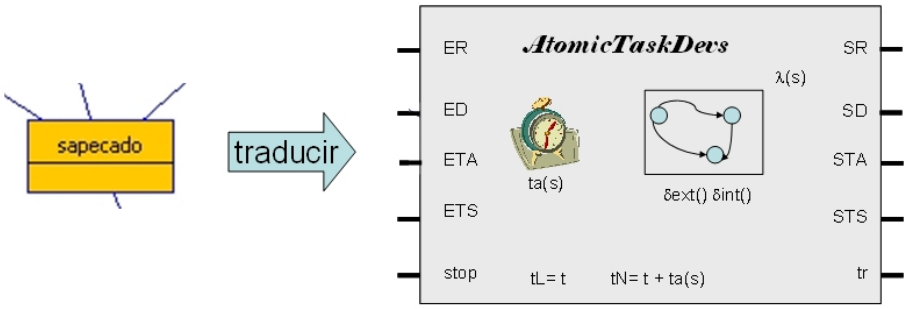


Figura 6.5: Transformación de una tarea en una AtomicTaskDevs

Transformation TaskToTaskDevs (Coordinate, DEVS)
Source

$$comp : Coordinates :: TaskImp \tag{6.34}$$

Target

$$modelD:\ DEVS::AtomicTaskDevs$$

$$SoloTareas : DEVS :: TaskVersionDevs \tag{6.35}$$

Source condition:

$$TaskImp.default.isEmpty() \tag{6.36}$$

Target condition:

$$modelD.ta(executing) = comp.execTime \tag{6.37}$$

$$SoloTareas.components \rightarrow includes(modelD) \tag{6.38}$$

$$modelD.name = comp.name \tag{6.39}$$

$$modelD.model = comp \tag{6.40}$$

$$modelD.inputports.name \rightarrow includes(\text{“}er\text{”}, \text{“}eta\text{”}, \text{“}ets\text{”}, \text{“}ed\text{”}, \text{“}stop\text{”}) \tag{6.41}$$

$$SoloTareas.outputport.name \rightarrow includes(\text{“}sr\text{”}, \text{“}sts\text{”}, \text{“}sta\text{”}, \text{“}sd\text{”}, \text{“}tr\text{”}) \tag{6.42}$$

Mapping

La Regla 3 establece la forma de transformar una tarea cuando ésta no tiene versión asociada, es decir es una tarea atómica. El modelo origen de la transformación es *comp* que representa una implementación de tarea según se especifica en la expresión 6.34. Como destino de la transformación se identifican dos modelos: *modelD* que representa una tarea atómica DEVS y *SoloTareas* que representa el modelo acoplado de mayor nivel, contenedor del modelo *modelD* generado (expresión 6.35). En la expresión 6.36 se establece como condición para transformar un modelo con esta regla (clase *TaskImp* en

la figura 6.2) que el mismo no tenga una versión asociada a través del vínculo *default*. Como condiciones de los modelos destinos, se establece en la expresión 6.37 que el tiempo en el estado *executing* de *modelD* es igual al tiempo de ejecución asociado a la tarea atómica *comp*. La condición 6.38 establece que *modelD* es uno de los componentes de *SoloTareas*. Finalmente en las expresiones 6.39 y 6.40 se establece que el atributo *name* del modelo destino es igual al atributo *name* del modelo origen y que existe una relación entre ambos modelos. No existen reglas de mapeo dado que se trata de un componente atómico que está completamente traducido.

Continuando con la transformación de los componentes de una versión descripta en la sección de mapeo en la Regla 2, la siguiente regla determina la conversión de un recurso cuando el mismo se encuentra como componente de una versión de tarea.

Regla 4: *Cada recurso (clase ResourceImp en la figura 6.2) que participa de la versión de tarea se transforma en un ResourceDevs*

El gráfico de la figura 6.6 describe la transformación de un recurso en un Resource-Devs.

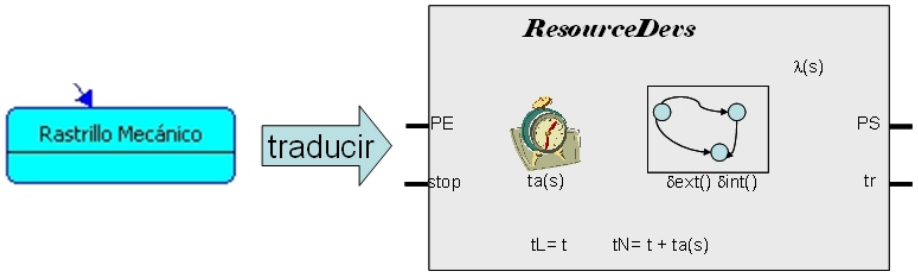


Figura 6.6: Transformación de un recurso en un ResourceDevs

Esta regla establece que por cada recurso que participa en la versión de tarea se genera una instancia de *ResourceDevs*. El modelo será acoplado con las tareas, una vez que el recurso forme parte del modelo de simulación, como se verá en la Regla 7. A

continuación se presenta la especificación de la regla.

Transformation ResourceToResourceDevs (Coordinate, DEVS)
Source

$$\begin{aligned} cvr &: Coordinates :: ResLifeCycle \\ comp &: Coordinates :: ResourceImp \end{aligned} \tag{6.43}$$

Target

$$\begin{aligned} modelD &: DEVS :: ResourceDevs \\ sm &: DEVS :: SimulationModel \end{aligned} \tag{6.44}$$

Source condition:

$$cvr = comp.class.theResLifeCycle \tag{6.45}$$

Target condition:

$$modelD.name = comp.name \tag{6.46}$$
$$modelD.resource = comp \tag{6.47}$$
$$modelD.inputports.name \rightarrow includes(\text{“}pe''\text{, “}stop'') \tag{6.48}$$
$$modelD.outputport.name \rightarrow includes(\text{“}tr''\text{, “}ps'') \tag{6.49}$$
$$sm.components \rightarrow includes(modelD) \tag{6.50}$$
$$modelD.state \rightarrow includes(cvr.theResState.type \rightarrow oclIsTypeOf(AtomicState).name) \tag{6.51}$$

$$\begin{aligned} modelD.deltext(e, \text{“}start'', s) &= s' : \\ cvr.theResState.theStateTransition &\rightarrow exists(t \,|t.origin = s \wedge t.destination = s' \wedge \\ t.startTrigger &\rightarrow notempty()) \end{aligned} \tag{6.52}$$

$$\begin{aligned} modeloD.deltext(e, \text{“}end'', s) &= dummy : \\ cvr.theResState.theStateTransition &\rightarrow exists(t \,|t.origin = s \wedge dummy \,\epsilon\, DUMMY \wedge \\ t.endTrigger &\rightarrow notEmpty() \end{aligned} \tag{6.53}$$

$$\begin{aligned} modeloD.deltint(dummy) &= s \\ cvr.theResState.theStateTransition &\rightarrow select(t \,|t.endTrigger.notEmpty() \wedge \\ t.origin = s\text{‘} &\wedge t.destination = s) \end{aligned} \tag{6.54}$$

Mapping

El modelo origen de la transformación es *ImpResource* perteneciente al modelo conceptual, y su ciclo de vida *cvr*, indicado por la expresión 6.43. Se identifican dos modelos destinos (expresión 6.44): *modelD* y *sm*. Se representa una restricción en el modelo origen, en donde se establece que el ciclo de vida *cvr* que se utiliza como fuente de la

transformación, es el ciclo de vida asociado al recurso que participa en el modelo conceptual, esto quiere decir que todos los recursos que son encontrados en el modelo conceptual pueden ser transformados por esta regla siempre que tengan asociado un ciclo de vida, de otra manera no podrán ser transformados. Las condiciones que se establecen en el modelo destino identifican ciertos valores de atributos del mismo y puertos de entrada y salida. La condición 6.46 indica que el atributo *name* del modelo destino tiene el mismo valor que el atributo *name* del modelo origen. La expresión 6.47 establece una relación entre los modelos origen y destino. Las condiciones 6.48 y 6.49 establecen los puertos de entrada y salida para el modelo destino de la transformación. La condición 6.50 establece la pertenencia de *modelD* como componente de *sm*. La condición 6.51 indica que todos los estados atómicos que se encuentran en el ciclo de vida del recurso en el modelo conceptual son también estados del modelo DEVS generado con esta regla.

Otro componente que puede ser encontrado en la versión de tarea que se transforma es una tarea que tenga asociada una versión por defecto. Se analizó en la Regla 3 la transformación de una tarea atómica, la Regla 5 trata la transformación de las tareas que se han dado en llamar compuestas, es decir aquellas que tienen asociada una descomposición en tareas más simples. Esta regla, presentada a continuación, es recursiva, dado que en la descomposición asociada a la tarea compuesta participarán otras tareas compuestas. Esta recursión se presenta en la sección de reglas de mapeo, donde se identifica la misma regla como regla de transformación de versiones de tareas. Por otro lado las tareas atómicas serán traducidas de acuerdo a la Regla 3 ya definida.

Regla 5: *Las versiones de tareas asociadas a tareas compuestas, se transforman en TaskVersionDevs*

La figura 6.7 describe la transformación de la tarea compuesta en una *TaskVersionDevs*, indicando que cada tarea que participa de la versión asociada a través del vínculo

default, es también transformada según la(s) regla(s) correspondiente participando como componente de la *TaskVersionDevs* que se genera.

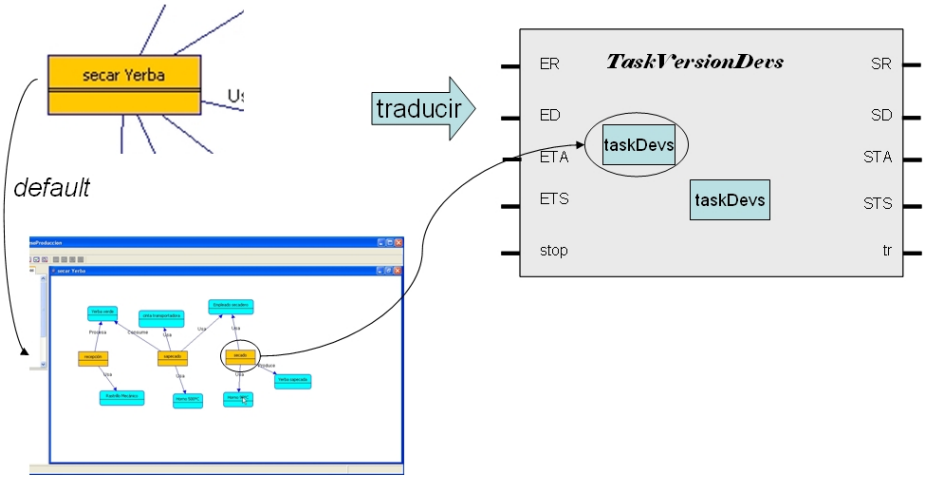


Figura 6.7: Transformación de una Versión de Tareas

A continuación se presenta la definición formal de la Regla 5.

Transformation VersionToVersionDevs(Coordinate, DEVS)
Source

$$comp : Coordinates :: TaskVersion \tag{6.55}$$

Target

$$\begin{aligned} modelD &: DEVS :: TaskVersionDevs \\ sm &: DEVS :: SimulationModel \\ firstTask &: DEVS :: TaskDevs \\ finalTask &: DEVS :: TaskDevs \end{aligned} \tag{6.56}$$

Source condition:
Target condition:

$$modelD.name = comp.name \tag{6.57}$$

$$modelD.inputports.name \rightarrow includes(\text{“}ed'', \text{“}eta'', \text{“}ets'', \text{“}stop'', \text{“}er'') \tag{6.58}$$

$$modelD.outports.name \rightarrow includes(\text{“}sd'', \text{“}sta'', \text{“}sts'', \text{“}sr'') \tag{6.59}$$

$$modelD.components \rightarrow includes(comp.components \rightarrow select(t\,|t.isOclTypeOf(TaskImp)).devs) \tag{6.60}$$

$$firstTask = comp.firstTask.devs \tag{6.61}$$

$$finalTask = comp.finalTask.devs \tag{6.62}$$

– *acoplamientos externos de entrada:*

$$modelD.cp \rightarrow exists(t\,|t.devs1 = modelD \wedge t.port1.name = \text{“}ed'' \wedge t.devs2 = firstTask \wedge t.port2.name = \text{“}ed'') \quad (6.63)$$

$$modelD.cp \rightarrow exists(t\,|t.devs1 = modelD \wedge t.port1.name = \text{“}ets'' \wedge t.devs2 = firstTask \wedge t.port2.name = \text{“}ets'') \quad (6.64)$$

$$modelD.cp \rightarrow exists(t\,|t.devs1 = modelD \wedge t.port1.name = \text{“}eta'' \wedge t.devs2 = firstTask \wedge t.port2.name = \text{“}eta'') \quad (6.65)$$

$$modelD.cp \rightarrow exists(t\,|t.devs1 = modelD \wedge t.port1.name = \text{“}er'' \wedge t.devs2 = firstTask \wedge t.port2.name = \text{“}er'') \quad (6.66)$$

$$modelD.cp \rightarrow exists(t\,|t.devs1 = modelD \wedge t.port1.name = \text{“}stop'' \wedge t.devs2 = firstTask \wedge t.port2.name = \text{“}stop'') \quad (6.67)$$

- - *acoplamientos externos de salida*

$$modelD.cp \rightarrow exists(t\,|t.devs1 = finalTask \wedge t.port1.name = \text{“}sd'' \wedge t.devs2 = modelD \wedge t.port2.name = \text{“}sd'') \quad (6.68)$$

$$modelD.cp \rightarrow exists(t\,|t.devs1 = finalTask \wedge t.port1.name = \text{“}sta'' \wedge t.devs2 = modelD \wedge t.port2.name = \text{“}sta'') \quad (6.69)$$

$$modelD.cp \rightarrow exists(t\,|t.devs1 = finalTask \wedge t.port1.name = \text{“}sts'' \wedge t.devs2 = modelD \wedge t.port2.name = \text{“}sts'') \quad (6.70)$$

$$modelD.cp \rightarrow exists(t\,|t.devs1 = finalTask \wedge t.port1.name = \text{“}sr'' \wedge t.devs2 = modelD \wedge t.port2.name = \text{“}sr'') \quad (6.71)$$

Mapping
try TaskToTaskDevs **on**

$$comp.components <\sim> modelD.components \quad (6.72)$$

try VersionToVersionDevs **on**

$$comp.components <\sim> modelD.components \quad (6.73)$$

try ResourceToResourceDevs **on**

$$comp.components <\sim> sm.components \quad (6.74)$$

try TemporalLinkToCoupling **on**

$$comp.components <\sim> modelD.cp \quad (6.75)$$

try TaskResLinkToCoupling **on**

$$comp.components <\sim> modelD.cp \tag{6.76}$$

En esta regla se establece la transformación de una versión de tarea y de cada uno de sus componentes. Inicialmente se identifican los modelos origen y destino de la transformación. La expresión 6.55 identifica al modelo fuente *comp* como una versión de tareas, es decir, una instancia de la clase *TaskVersion*. Luego la expresión 6.56 identifica los modelos destinos: *modelD, sm, firstTask* y *finalTask*. No se establecen restricciones para el modelo origen, indicando que toda versión de tarea puede ser transformada aplicando esta regla. Las condiciones para los modelos destinos, establecen valores de atributos, acoplamientos y componentes. La expresión 6.57 establece que el valor del atributo *name* del modelo destino es igual al valor del atributo *name* del modelo origen. Las condiciones 6.58 y 6.59 establecen los puertos de entrada y salida del modelo destino. La expresión 6.60, identifica los componentes del modelo destino como los modelos DEVS asociados a los componentes del modelo origen. Las dos condiciones que siguen, 6.61 y 6.62 identifican dos componentes del modelo destino llamadas *firstTask* y *finalTask* que corresponden a los modelos DEVS asociados con las componentes del modelo origen que pertenecen a los conjuntos *firstTask* y *finalTask* definidos en 5.23 y 5.27 respectivamente. Estos modelos se identifican para poder definir los acoplamientos externos de entrada y acoplamientos externos de salida. Las condiciones 6.63 a 6.67 identifican los acoplamientos externos de entrada indicando que los puertos de entrada del modelo *modelD* están acoplados con los puertos de entrada del mismo nombre de las tareas identificadas como *firstTask*. Y las condiciones 6.68 a 6.71 identifican los acoplamientos externos de salida, donde los puertos de salida de los modelos *finalTask* están acoplados con los puertos del mismo nombre del modelo *modelD*.

Las dos reglas siguientes determinan cómo las relaciones temporales y relaciones tareas-recursos generan los acoplamientos entre los modelos DEVS.

Regla 6: *Una relación temporal se transforma en un acoplamiento entre los modelos DEVS asociados a las tarea que son origen y destino de esta relación.*

Transformation TemporalLinkToCoupling(Coordinates, DEVS)
Source

$$tl : Coordinates :: TemporalLink \tag{6.77}$$

Target

$$\begin{aligned} coup &: DEVS :: couprel \\ taskDsourc &: DEVS :: TaskDevs \\ taskDend &: DEVS :: TaskDevs \end{aligned} \tag{6.78}$$

Source condition:
Target condition:

$$taskDsourc = tl.source.devs \tag{6.79}$$
$$taskDend = tl.end.devs \tag{6.80}$$
$$portSourc.name = \text{“}sta\text{”} \tag{6.81}$$
$$portEnd.name = \text{“}et\text{”} \tag{6.82}$$

$$\begin{aligned} tl &\rightarrow oclIsTypeOf(Before \vee During \vee Meet)\ implies \\ taskDsourc.container.cp &\rightarrow exists(p\,|p.devs1 = taskDsourc \wedge p.port1.name = \text{“}sta\text{”} \wedge \\ &\quad p.devs2 = taskDend \wedge p.port2.name = \text{“}eta\text{”}) \end{aligned} \tag{6.83}$$

$$\begin{aligned} tl &\rightarrow oclIsTypeOf(Meet \vee Equal \vee Begin)\ implies \\ taskDsourc.container.cp &\rightarrow exists(p\,|p.devs1 = taskDsourc \wedge p.port1.name = \text{“}sts\text{”} \wedge \\ &\quad p.devs2 = taskDend \wedge p.port2.name = \text{“}ets\text{”}) \end{aligned} \tag{6.84}$$

Mapping

Esta regla indica la forma de establecer los acoplamientos entre las tareas que son origen y destino de las relaciones temporales dependiendo del tipo de la relación. Así los puertos de las relaciones identificadas como síncronas y asíncronas se acoplan. Como se observa, no existen reglas de mapeo dado que este elemento no tiene otras reglas de transformación asociadas.

Las relaciones tareas-recursos que se encuentran en los modelos DEVS de origen, no se transforman directamente en un acoplamiento entre una tarea DEVS y un recurso DEVS. Esto se debe a la estructura que se plantea para el modelo de simulación donde

los recursos son representados en el modelo de mayor nivel y no dentro de las versiones de tareas como es el caso de los modelos Coordinates. En este sentido, los acoplamientos se dan entre los puertos de las tareas con sus contenedores, es decir, los modelos acoplados del que forman parte. De esta forma los eventos provenientes de los recursos se propagan de modelo en modelo desde el más externo al más interno.

Regla 7: *Una relación tarea-recurso del modelo Coordinates se transforma en un acoplamiento externo de entrada y de salida entre los modelos DEVS componentes de una versión y los puertos de entrada y salida de dicha versión*

Transformation TaskResLinkToCoupling (Coordinates, DEVS)
Source

$$relTR : Coordinates :: TaskResLinkImp \tag{6.85}$$

Target

$$cp : DEVS :: couprel \tag{6.86}$$

Source condition:
Target condition:

$$taskDsource = relTR.source.devs \tag{6.87}$$

$$taskDen = relTR.end.devs \tag{6.88}$$

$$taskDsourc.container.cp \rightarrow exists\,(p\,|p.devs1 = taskDsource \wedge p.port1.name = \text{“}sr'' \wedge p.devs2 = taskDsourc.container \wedge p.port2.name = \text{“}sr'') \tag{6.89}$$

$$taskDsource.container.cp \rightarrow exists\,(p\,|p.devs1 = taskDsource.container \wedge p.port1.name = \text{“}er'' \wedge p.devs2 = taskDsource \wedge p.port2.name = \text{“}er'') \tag{6.90}$$

$$relTR \rightarrow oclIsTypeOf(process)\ implies$$
$$taskDsourc.container.cp \rightarrow exists\,(p\,|p.devs1 = taskDsourc.container \wedge p.port1.name = \text{“}ed'' \wedge p.devs2 = taskDsourc \wedge p.port2.name = \text{“}ed'') \tag{6.91}$$

Mapping

Esta regla establece la transformación de las relaciones entre tareas y recursos existentes en el modelo conceptual. Así la expresión 6.85 identifica el origen de la transformación como una relación tarea-recurso. Como resultado de la transformación, la expresión 6.86 establece la generación de relaciones de acoplamientos entre modelos DEVS.

Las expresiones 6.87 y 6.88 identifican a los modelos origen y destinos de las relaciones de acoplamientos que serán generadas. La expresión 6.89 establece el acoplamiento externo de salida entre el modelo de la tarea que es origen de la relación que se está transformando con el modelo contenedor de la misma. Así, esta expresión establece el acoplamiento entre el puerto *sr* de la tarea origen con el puerto *sr* de su contenedor. La expresión 6.90 establece el acoplamiento externo de entrada, indicando que el puerto *er* del modelo contenedor está acoplado con el puerto *er* de la tarea origen. En el caso particular que la relación entre tarea y recurso sea del tipo *process*, entonces los puertos *ed* del contenedor y la tarea también estarán asociados como se describe en la expresión 6.91.

6.2. Capa de Simulación

La especificación presentada en el punto anterior se implementa en la capa de simulación (Simulation layer) La capa de simulación tiene como principales funcionalidades la traducción del modelo conceptual de empresa (EM) en un modelo de simulación (SM) y su ejecución tanto en entorno local como distribuido.

Para el desarrollo de esta capa se utilizó el patrón de diseño MSVC (Nutaro y Hammonds, 2004) que propone una clara separación entre el modelo de simulación, el intérprete del modelo y el controlador de la simulación. La figura 6.8 muestra la arquitectura de la capa de simulación, la cual consta de tres componentes: *Enterprise Simulation model*, *Coordinator* y *Simulator*. El primero de éstos, *Enterprise Simulation Model*, comprende la definición de los bloques de construcción del modelo de simulación. Éstos, son implementaciones de los modelos conceptuales DEVS definidos en las secciones 5.1 a 5.5. Forma parte de *Enterprise Simulation Model* el componente *Translator* encargado de implementar las reglas definidas en 6.1

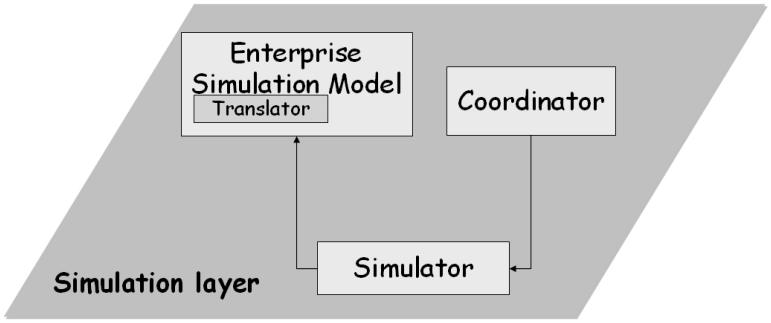


Figura 6.8: Arquitectura de la capa de simulación

El componente *Coordinator* se encarga de ejecutar el SM en ambiente local y distribuido, proveyendo los mecanismos necesarios para tal fin. Este componente permite controlar la simulación implementando el ciclo de simulación correspondiente, dirigiendo y controlando los elementos que intervienen, el envío y recepción de interacciones y el avance del tiempo.

El componente *Simulator* contiene las máquinas de simulación que interpretan los modelos atómicos y acoplados que componen el SM.

En este sentido la capa de simulación recibe desde la capa superior el EM y retorna como servicio los resultados de su ejecución. Así, esta capa es responsable de solucionar los problemas de traducción de modelos, de ejecución y de conexión cuando se trata de simulación distribuida. El gráfico de la figura 6.9 muestra el diagrama de casos de uso de esta capa. En el desarrollo de la misma, se utilizó el framework DEVSJAVA V3.1 (ACIMS, 2004) que implementa el formalismo DEVS en el lenguaje de programación Java. Este formalismo emplea los conceptos del paradigma de objetos (Sarma y otros, 1998) para la construcción de sus modelos (modularidad y abstracción), al igual que lo hace Coordinates.

Para la traducción del modelo de empresa en el modelo de simulación, se parte de la *Vista de tareas* (Modelo Coordinates), es decir, aquella que representa la vista

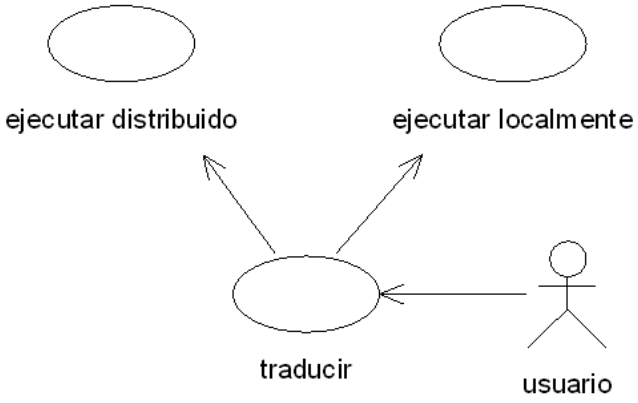


Figura 6.9: Diagrama de caso de usos de la capa de simulación

funcional de la organización, sin embargo no se dejan de lado las otras vistas. Así, en la transformación de los modelos de tareas, los recursos que intervienen son traducidos en modelos de simulación de acuerdo al ciclo de vida asociado al mismo, como se describió en la Regla 4. De esta manera el modelo de simulación que se obtiene refleja la vista integrada de Empresa y no solo su aspecto funcional. A continuación, en las próximas secciones se detallan los componentes de la arquitectura que dan soporte a las funcionalidades definidas para esta capa.

6.2.1. Transformación del Modelo Conceptual de Empresa (EM)

Los componentes que dan soporte al proceso de traducción de modelos se encuentran definidos en el componente *Enterprise Simulation Model* de la capa de simulación los cuales pueden ser visualizados en la figura 6.10.

Las clases *AtomicTaskDevs, TaskVersionDevs, ResourceDevs, SimulationModel* y *ExperimentalFrame* corresponden a las implementaciones de los modelos definidos formalmente en 5.1 a 5.5.

En la implementación de *AtomicTaskDevs* se utilizó el patrón de diseño *State* para

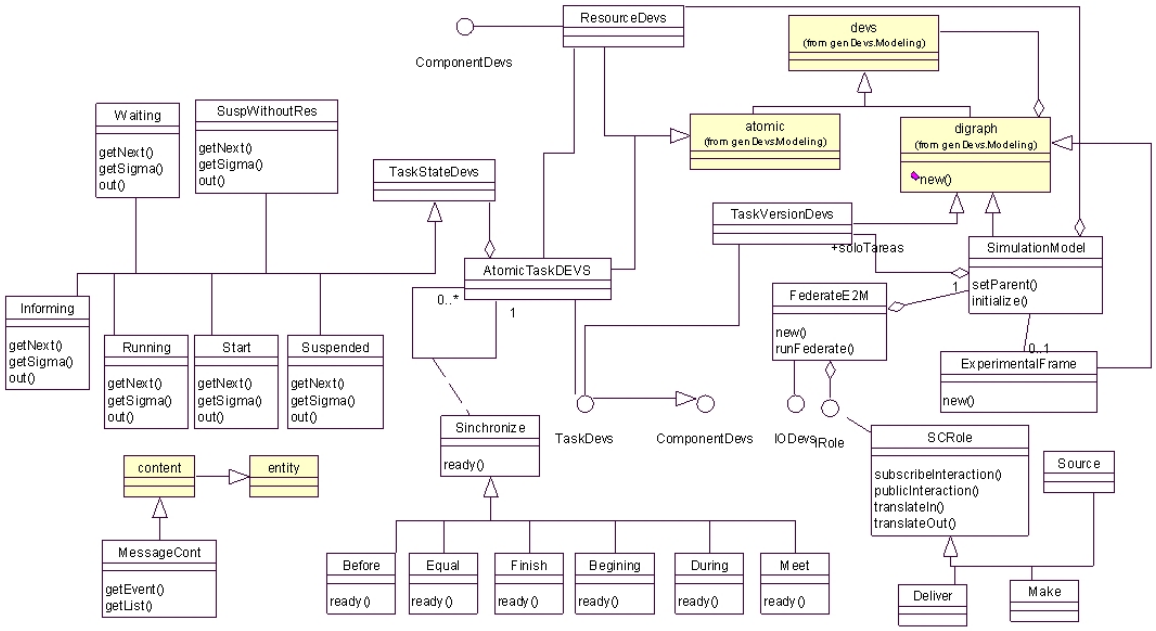


Figura 6.10: Componentes del Enterprise Simulation Model

representar los distintos estados en que el modelo de simulación, correspondiente a una tarea atómica, puede encontrarse. Estos estados implementan las funciones de transición interna y externa definidas en el modelo conceptual. Dado que las funciones de salida (out) y del tiempo (getSigma) también dependen del estado en que el modelo se encuentra, las mismas fueron definidas en éste. La clase *Sinchronize* permite simplificar la lógica del modelo de simulación de una tarea, ya que se encarga de controlar cuándo la tarea se encuentra lista para ejecutar dependiendo de las relaciones temporales con las otras tareas. De esta forma, cada vez que una *AtomicTaskDevs* recibe un evento por los puertos *ETA* o *ETS* (los eventos posibles son *ready, start* y *end*) delega el tratamiento de los mismos en las subclases de *Sinchronize*.

En lo referido al empleo de los modelos de simulación en el contexto distribuido, la clase *FederateE2M* implementa el federado que puede actuar en el mismo como se describió en el punto 3.1.2. Este modelo tiene como componentes al modelo de simulación

(clase *SimulationModel*) y un rol (clase *IRol*) que determina el conjunto de interacciones que el federado genera y aquellas en las cuales está interesado en recibir. La clase *SCRole* representa los posibles roles que el federado puede tener asociado. Esta clase implementa la interfaz *IRole* y fue definida como un ejemplo de una clase concreta para demostrar su efectividad en lograr los objetivos que se plantearon en cuanto a la interoperabilidad semántica. Los principales servicios definidos en la interfaz son: *subscribeInteraction*, *publishInteraction*, *translateIn* y *translateOut*.

Los métodos *subscribeInteraction*y *publishInteraction* son los servicios necesarios para subscribir y publicar interacciones para el federado. Estos métodos invocan los servicios *subscribeInteraction* y *publishInteraction* respectivamente sobre el *RTIAmbassador* (esta clase implementa el EmbajadorRTI explicado en 2.2.2 y es parte del framework *Portico* utilizado en esta tesis). De esta manera el rol conoce cuales interacciones deben ser subscriptas y publicadas para el federado, el federado delega en el rol estas acciones. Los métodos *translateIn* y *translateOut* corresponde a la transformación de las interacciones, que llegan desde otros federados, en eventos que puedan ser interpretados por los modelos DEVS y vice versa. La clase *SCRole* tiene tres subclases que representan los roles concretos: *Source*, *Make* y *Deliver*. Estos roles tienen como objetivo proveer interoperabilidad semántica a los federados que interactúan en una federación. La definición de los roles se hizo basado en el modelo SCOR (Harmon, 2003) para cadenas de suministro. Así el rol *source* contiene interacciones para obtener mercadería y/o servicios, el rol *Make* contiene interacciones que transforman la materia prima en productos elaborados. Finalmente el rol *deliver* incluye interacciones para proveer productos intermedios o finales y/o servicios. Todos los roles incluyen interacciones que corresponden a los procesos de *return* definidos en SCOR que permiten retornar mercadería en mal estado o en exceso.

Hasta ahora se han definido roles en el contexto de cadenas de suministros pero otros

roles pueden ser definidos siguiendo estos criterios. La utilización de roles permite a un federado variar las interacciones que envía y recibe dinámicamente. Así un federado puede participar en diferentes cadenas de suministro con diferentes roles, pudiendo reusar el federado sin necesidad de re-programación.

El primer paso en la ejecución del modelo de empresa, es la transformación del mismo en un modelo de simulación que pueda ser interpretado por un simulador. Así, la capa de simulación provee los elementos necesarios para dar soporte a estas operaciones. El proceso de transformación es descripto en el diagrama de secuencia de la figura 6.11.

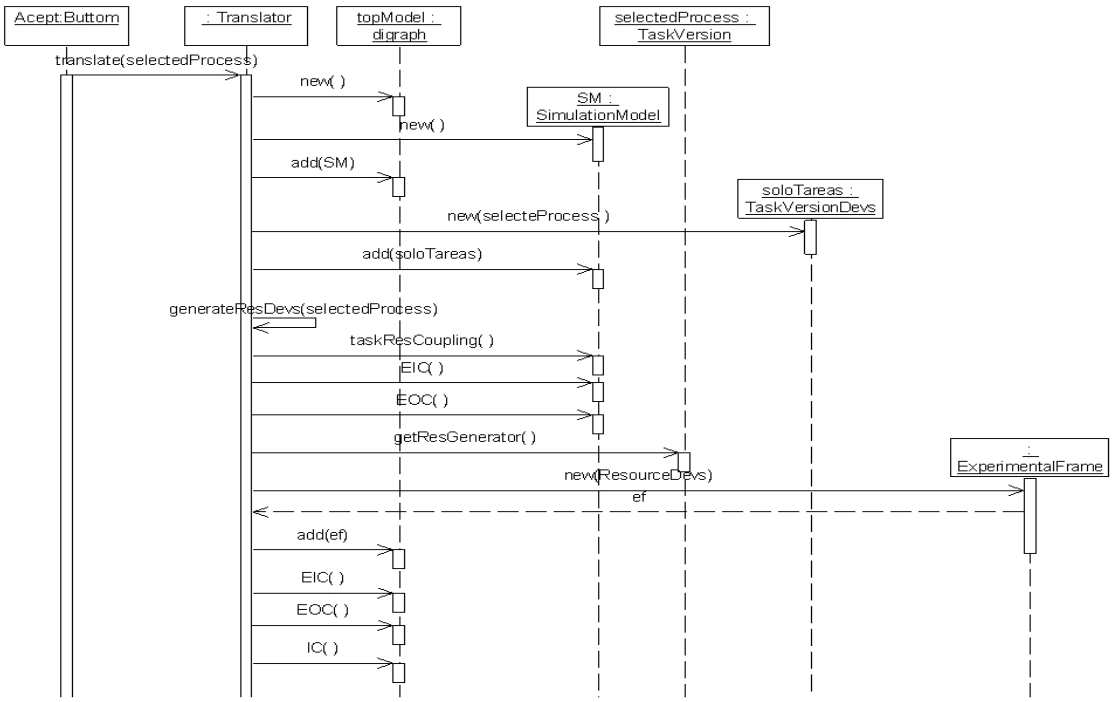


Figura 6.11: Diagrama de secuencia para la traducción de modelos

En la figura 6.11 la traducción del modelo se inicia con el envío del mensaje *translate(selectedProcess)*. A partir del proceso seleccionado (*selectedProcess*), se crea el modelo de mayor nivel, referenciado en la figura como *topModel*, el cual a su vez tiene dos modelos componentes: una instancia de *SimulationModel* y una instancia de *Experi-*

mentalFrame. *TopModel* es un digrafo, es decir un modelo DEVS acoplado que tiene como componentes, en este caso particular, dos modelos DEVS acoplados. A partir del proceso seleccionado, se genera el modelo *soloTareas* el cual es un modelo acoplado que representa la descomposición de la tarea proceso seleccionada, en subtareas, dicho modelo es una instancia de *TaskVersionDevs*. Este modelo se agrega como componente de *SimulationModel*. Una vez generado el mismo, se crean las instancias de los modelos que representan los recursos. El método que se invoca *generateResDevs* tiene como finalidad crear las instancias de *ResourceDevs* correspondientes a los recursos que participan en la versión seleccionada. Estos modelos generados se agregan como componentes del *SimulationModel* a través del método *add*. Una vez definidos los componentes de *SimulationModel*, se generan los acoplamientos entre ellos. El método *taskResCoupling* invocado, se encarga de generar los acoplamientos entre los modelos de los recursos, miembros de *SimulationModel* y el modelo *soloTareas*, es decir, este método corresponde a la definición del acoplamiento interno (IC) en el modelo *SimulationModel*. A continuación, se invocan los métodos *EIC* y *EOC* correspondientes a los acoplamientos externos de entrada y acoplamientos externos de salida respectivamente. Luego, se crea el marco experimental (ef) el cual se agrega como componente del modelo de mayor nivel. Finalmente, se invocan los métodos encargados de generar los acoplamientos en *topModel* (IC, EIC y EOC).

6.2.2. Ejecución Local del Modelo de Simulación

Para ejecutar el modelo de simulación que se genera, es necesario asociar a los modelos, los intérpretes de los mismos y el ciclo de simulación. Estos elementos están representados en los componentes *Simulator* y *Coordinator* de esta capa (6.8).

El componente *Simulator* contiene las máquinas que interpretan los diferentes mo-

delos. Dado que los componentes del modelo de simulación que se generan a través del uso de esta arquitectura son modelos DEVS, se utilizaron las clases definidas en el paquete *gendevs.simulation*((ACIMS, 2004)) para interpretar el modelo de simulación y sus componentes. Estas clases son las máquinas de simulación apropiadas para poder interpretar los modelos DEVS definidos para el caso especial de los modelos de empresa. La figura 6.12 muestra el diagrama de clases del componente *Simulator*.

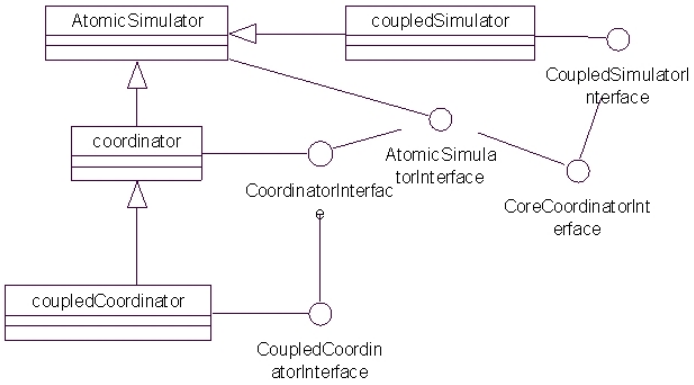


Figura 6.12: Diagrama de Clases del componente Simulator

De esta forma, un modelo DEVS atómico que forma parte de modelos DEVS acoplados, tendrá asociado una instancia de *coupledSimulator*. Un modelo DEVS acoplado puede formar parte de otro modelo DEVS acoplado, y en este caso tendrá asociado un *coupledCoordinator*. Otra alternativa es que un modelo DEVS acoplado no forme parte de otro modelo DEVS acoplado, en este caso será un modelo del más alto nivel en la jerarquía, por lo cual tendrá asociado un *coordinator*, el cual implementa el ciclo de simulación y corresponde al coordinador raíz.

El esquema de la figura 6.13 muestra un ejemplo de la jerarquía de simuladores, coordinadores y las relaciones que se establecen a partir del modelo de simulación que produce la transformación del modelo Coordinates.

Como se observa, el modelo de mayor nivel, indicado en la figura como *modeloSu-*

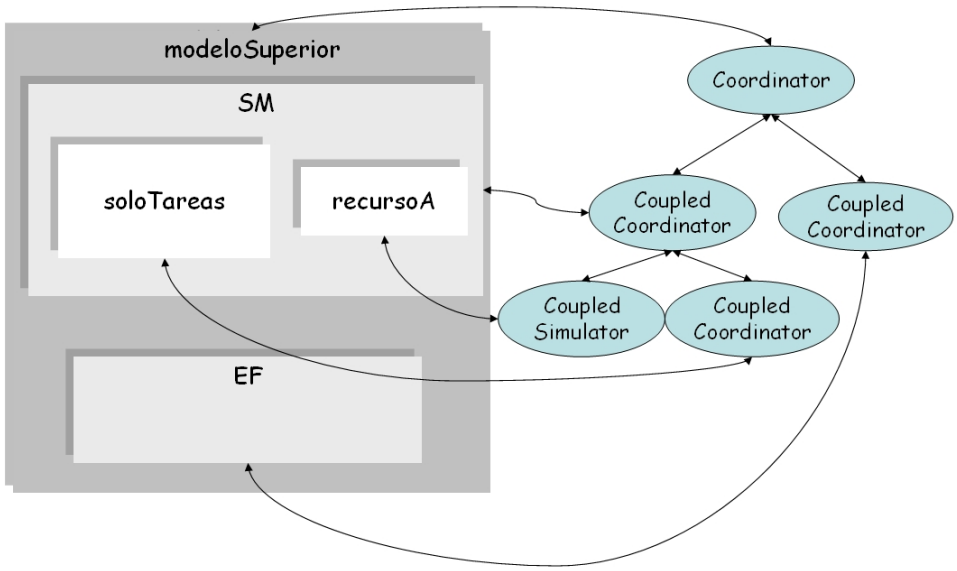


Figura 6.13: Relación modelos DEVS e Intérpretes

perior tiene asociado un *coordinator* que representa el coordinador raíz. Este modelo tiene dos componentes: *SM* y *EF* correspondientes ambos, a modelos acoplados (representando el modelo de simulación y su marco experimental), cada uno de dichos modelos tiene asociado un *coupledCoordinator*. El modelo *SM* esta compuesto por dos submodelos: *soloTareas* (correspondiente a un modelo acoplado) y *recursoA* (correspondiente a un modelo atómico). Estos modelos tienen asociado un *coupledCoordinator* y un *coupledSimulator* respectivamente. Los coordinadores y simuladores forman una jerarquía idéntica a la jerarquía de los modelos a los que se encuentra asociados.

El coordinadore raíz implementa el ciclo de simulación DEVS, el cual se presenta en la figura 6.14. El ciclo calcula el tiempo del próximo evento (TN), luego se calculan las entradas y salidas, se envían los mensajes y finalmente se ejecuta la función de transición.

Para presentar más claramente en el ciclo de simulación la interacción entre los simuladores y coordinadores, se presenta en la figura 6.15 un ejemplo de un modelo

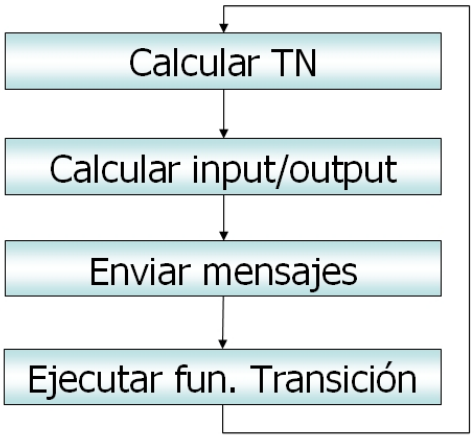


Figura 6.14: Ciclo de simulación DEVS

acoplado AB con dos modelos atómicos componentes: el modelo A y el modelo B. Cada uno de los modelos tiene asociado coordinadores y simuladores que forman una jerarquía. El ciclo comienza cuando el coordinador envía la solicitud de obtener el tiempo del próximo evento (tN) a sus simuladores asociados enviando el mensaje *nextTN*. Los simuladores responden con el valor de tN en el mensaje *outTN* enviado al coordinador. Luego el coordinador envía a cada simulador el mensaje *getOut*, que contiene el valor *tN* global. Cada simulador analizará si corresponde o no enviar las salidas, si el simulador es inminente, es decir, su valor de *tN* es igual al valor de *tN* global, invoca el método *out* en su modelo asociado. Las salidas producidas por los modelos son enviadas al coordinador en el mensaje *sendOut*. El coordinador utiliza los acoplamientos especificados para distribuir las salidas como mensajes DEVS[1] a sus simuladores a través del mensaje *applyDelt*. Para aquellos simuladores que no tienen entradas en sus puertos, el mensaje enviado está vacío. Cuando los simuladores reciben el mensaje *applyDelt* deben actuar de la siguiente manera:

[1] aquí se está haciendo referencia a los mensajes intercambiados entre los modelos de simulación y no a mensajes enviados entre objetos

si tienen mensajes de entradas en sus puertos y son inminentes deben invocar el método *deltcon* en sus modelos

si tienen mensajes de entradas en sus puertos pero no son inminentes deben invocar el método *deltext* en sus modelos

si no tienen mensajes de entradas en sus puertos pero son inminentes deben invocar el método *deltint* en sus modelos

si no tienen mensajes de entradas en sus puertos y no son inminentes no hacer nada.

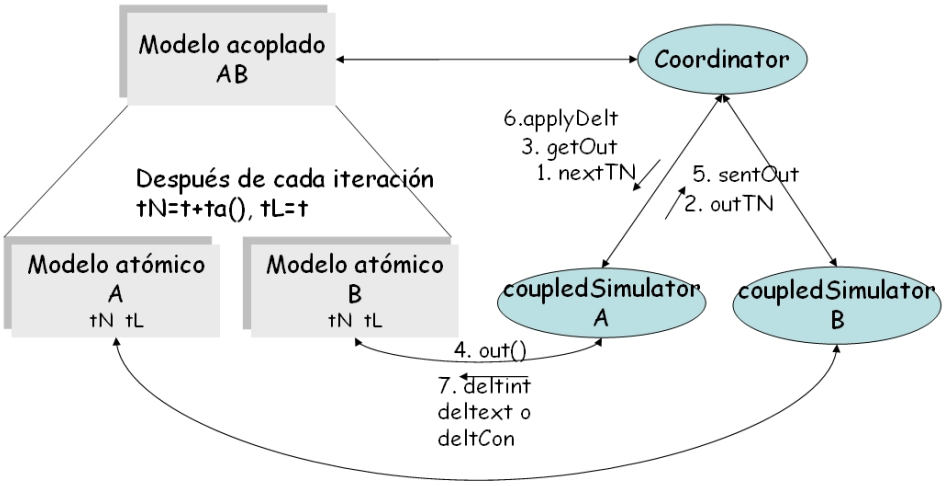


Figura 6.15: Ejemplo del funcionamiento del ciclo de simulación

6.2.3. Ejecución Distribuida del Modelo de Simulación

Para la ejecución distribuida de este modelo, se genera un federado HLA, capaz de interactuar con otros federados a través de un entorno de ejecución o RTI (Run Time Infrastructure). Este federado debe tener como simulador al modelo DEVS generado

por la capa de simulación. La figura 6.16 muestra el diagrama de secuencias que se ejecuta cuando se requiere simular el modelo de empresa en un entorno distribuido.

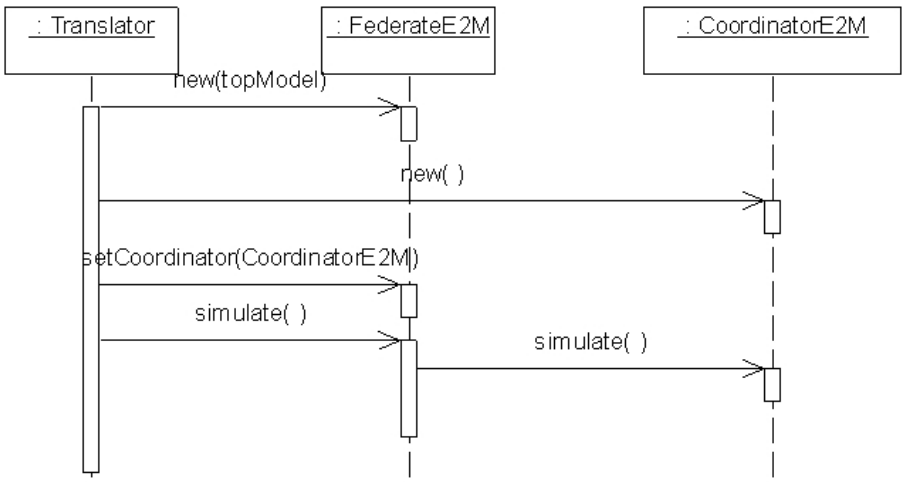


Figura 6.16: Diagrama de secuencias simulación distribuida

Se observa que el traductor (:Translator), crea una instancia de *FederateE2M* y le asocia como coordinador una instancia de *CoordinatorE2M*, luego, invoca el método *simulate* para comenzar la simulación distribuida.

En este diagrama intervienen dos conceptos nuevos: *FederateE2M* y *CoordinatorE2M*. Estas clases se encargan de dar soporte a la ejecución en un entorno distribuido del modelo de simulación. *FederateE2M* representa el federado HLA cuyo simulador está representado por el digrafo *topModel* pasado como argumento en el método *new()*. *CoordinatorE2M* es el intérprete asociado a *FederateE2M* que adapta el ciclo de simulación DEVS al entorno distribuido. A continuación se describen de que forma participan en la simulación distribuida las clases *CoordinatesE2M* y *FederateE2M*.

6.2.3.1. CoordinatorE2M

En la ejecución de la simulación local se utilizó la clase *coordinator* provista por el framework DEVSJAVA. Esta clase, implementa el ciclo de simulación DEVS y permite controlar la simulación correctamente en el entorno local, pero en el entorno distribuido, este esquema no funciona.

El principal problema que surge es la administración del tiempo. DEVS implementa un esquema de manejo del tiempo optimista libre de riesgos, esto es, no contempla la existencia de eventos *straggles* (eventos que arriban con un tiempo menor que el tiempo actual de simulación, descriptos en 2.2.1) y por lo tanto no se implementa un mecanismo de *rollback* para poder volver atrás en el tiempo de simulación ante posibles inconvenientes con la aparición de este tipo de eventos.

Como fue explicado en el capítulo 2, sección 2.2.1, existen dos formas genéricas propuestas de administración del tiempo, los cuales corresponden al esquema conservativo y al esquema optimista. Ambos esquemas pueden ser vistos como avanzando el tiempo hacia delante mientras procesan eventos con marcas temporales. La primera de estas elimina la violación a la causalidad estrictamente, esto es, en este tipo de esquema, siempre la causa aparece antes que el efecto. En cambio, en el segundo esquema se permite temporalmente la violación a la causalidad, pero esta es detectada y corregida a través del uso de *rollback*. Un tercer esquema es el protocolo DEVS paralelo, el cual puede ser visto como un esquema optimista libre de riesgos, donde el *rollback* no es implementado en los componentes (Zeigler y otros, 1998). En este caso, cuando un componente recibe un mensaje que corresponde a su pasado, informa sobre una mala sincronización, pero no la corrige. El coordinador raíz es el encargado de sincronizar los elementos como se explicó en el punto anterior.

Desde la perspectiva de HLA, el estándar permite que cada federado seleccione la

forma de administrar sus eventos indicando cómo enviar y recibir los mismos (a través de las funciones *timeConstrain* y *timeRegulated*). En este esquema, el federado es responsable por administrar su tiempo de simulación interno y de mantener su sincronización con los otros federados. La tabla 6.1 muestra una comparación entre el protocolo DEVS paralelo y el estándar HLA de simulación distribuida.

Característica	DEVS	HLA
Admin. del tiempo	optimista s/Roll back	optimista o conservativo
Comunicación	mensajes	interacciones y actualización de atributos
Avance del tiempo y coordinación	coordinador raíz	RTI
Envío de mensajes	controlado por Coordinador raíz	esquema publicar/subscribir

Tabla 6.1: Comparación HLA y Protocolo DEVS

En principio, el federado HLA cuyo simulador es un modelo DEVS (llamado de aquí en adelante federado DEVS), podría ejecutar sin inconvenientes, dado que HLA permite el esquema optimista para el manejo del tiempo. Pero el problema se presenta en el ciclo de simulación DEVS que no implementa *rollback*, lo cual provocaría errores irrecuperables ante la existencia de eventos straggles. La forma de solucionar esto es modificando de alguna manera el esquema de simulación DEVS. Existen dos alternativas: implementar *rollback* o implementar un esquema conservativo.

En esta tesis se optó por implementar un esquema conservativo. Para ello, cuando el federado DEVS necesite avanzar el tiempo, debe solicitar permiso para hacerlo al RTI, de esta manera el RTI y el federado DEVS garantizan no enviar eventos en el pasado. Bajo este análisis es necesario modificar el algoritmo de simulación DEVS de manera que incorpore dicho cambio.

Para ello se define un nuevo coordinador raíz encargado de dirigir la simulación en un entorno distribuido. Este coordinador llamado *CoordinatorE2M* redefine el método

simulate con el objetivo de resolver el problema planteado. Durante el ciclo normal de simulación, el coordinador raíz, *coordinatorE2M*, calcula el tiempo del próximo evento y, antes de avanzar a la próxima etapa, solicita al RTI permiso para avanzar el tiempo, invocando el servicio *nextEventRequest(tn)*. La figura 6.17 esquematiza la secuencia de tareas que implementan el método *simulate* modificado.

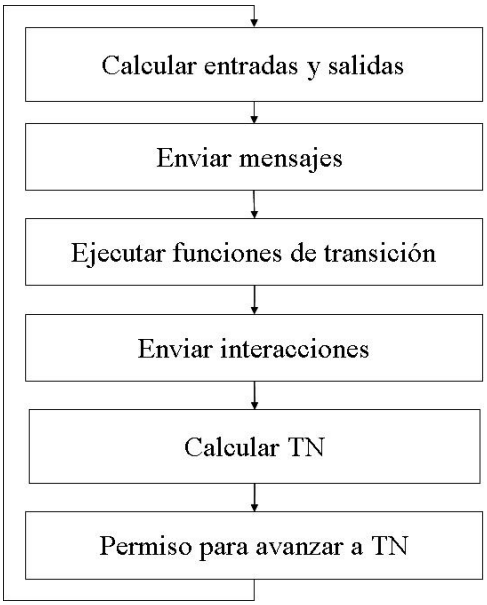


Figura 6.17: Método Simulate Modificado

Un punto importante a tener en cuenta con los federados conservativos es el concepto *lookahead* (Fujimoto, 1988). Si se analiza el modelo de simulación generado usando los modelos DEVS, presentados en el capítulo anterior, es posible notar que dichos modelos tienen un *lookahead* igual a cero. Esto es, si se quisiera determinar cuál es el tiempo más pequeños en que un evento puede aparecer, éste es cero dado que el modelo tiene estados llamados ficticios (dummy) de duración cero. Este hecho puede traer problemas en la entrega de las interacciones hacia otros federados. HLA 1516 a diferencia de su predecesora, permite que un federado tenga este valor en cero, pero una vez que el federado recibe el permiso para avanzar al tiempo del próximo evento *tN*, tiene

prohibido enviar eventos con un tiempo igual a *tN*. Luego, para eliminar este problema, el *coordinatorE2M* antes de enviar la solicitud para avanzar el tiempo debe verificar que no haya componentes inminentes en el modelo de simulación con tiempo *tN*. Si esto sucediera, tendrá que solicitar a dicho componente el envío de los mensajes de salida. Una vez identificados todos los mensajes de salida para el tiempo *tN*, entonces se está en condiciones de solicitar al RTI el permiso para avanzar el tiempo, garantizando de esta manera no enviar eventos con tiempos menores o iguales a *tN*.

La figura 6.18 muestra el diagrama de colaboraciones entre los distintos componentes: modelos, intérpretes de estos modelos y coordinadores para el caso de la administración del tiempo.

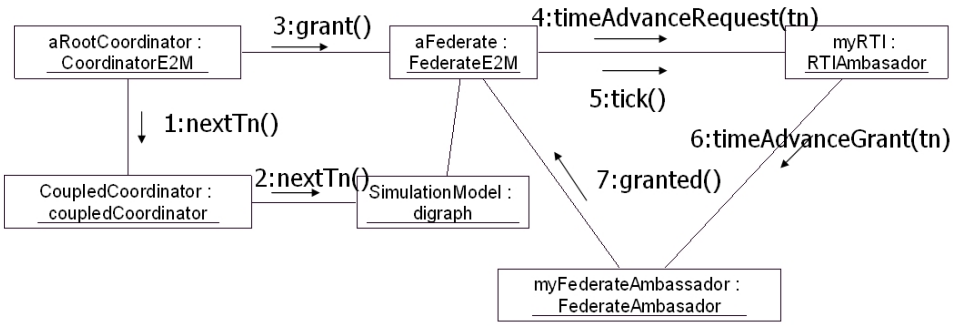


Figura 6.18: Diagrama de Colaboraciones para administración el tiempo

El objeto *aRootCoordinator*, una instancia de *CoordinatorE2M*, es responsable de dirigir la ejecución de la simulación en el entorno distribuido. El objeto *aFederate* (una instancia de FederateE2M) es responsable de interactuar con otros federados a través de la interacción con el *RTIAmbassador* y el *FederateAmbassador* para poder alcanzar sus objetivos. El *SimulationModel* representa el comportamiento de la empresa y al ser un digrafo, tiene un *coupledCoordinator* asociado.

En este escenario, *aRootCoordinator* solicita el tiempo del próximo evento al *coupleCoordinator* asociado en la jerarquía de simuladores y coordinadores. Este mensaje es

propagado en la jerarquía de los intérpretes de los modelos DEVS. Una vez que el elemento *aRootCoordinator* tiene el valor *tN*, antes de avanzar el tiempo de simulación, debe solicitar el permiso al RTI, para hacerlo, delega esta responsabilidad en el objeto *aFederate*. Para llevar a cabo este servicio, *aFederate* invoca el servicio *timeAdvanceRequest* al *RTIAmbassador* y espera a recibir la respuesta. Pasados unos segundos, una instancia de *RTIAmbasador*, *myRTI* envía el permiso invocando la función *timeAdvanceGrant* sobre *myFederateAmbassador*, una instancia de *FederateAmbassador*. Inmediatamente, informa de la recepción del permiso al federado quien avisa al coordinador raíz. Finalmente, el coordinador raíz puede continuar con su ciclo de simulación.

6.2.3.2. Definición del Federado FederateE2M

El simulador que se define en esta sección representa el comportamiento del modelo de empresa capaz de interactuar en un entorno distribuido bajo la especificación HLA, es decir es un federado HLA. En este contexto, se ha desarrollado un modelo especial llamado **FederateE2M** el cual cumple la especificación HLA.

Ser un federado HLA, significa cumplir la definición de federado dada por la especificación IEEE (2000b): un federado es *una aplicación que puede ser, o está actualmente acoplado, con otras aplicaciones de software bajo los datos de documentación del modelo de objetos de la federación (FDD) y la infraestructura de ejecución (RTI)*. En otras palabras, un federado HLA, debe ser capaz por un lado, de reconocer e interpretar los datos de intercambio con otros federados que tienen un formato establecido por el estándar HLA, este formato es un archivo XML (XML, 2008) con características propias que representa el FOM. Por otro lado, el federado debe poder interactuar con otros federados a través de una infraestructura de ejecución definida por el estándar, llamada RTI (Run Time Infrastructure). La especificación no proporciona la implementación para RTI pero sí la descripción del conjunto de servicios que el mismo debe proveer.

De esta manera, un federado HLA es capaz de hacer uso de esos servicios y de prestar servicios al RTI. Los servicios que el federado presta al RTI son llamados *call back function* en la especificación. Existen una gran variedad de implementaciones de RTI que cumplen la especificación y que tienen certificado del organismo de estandarización IEEE, la mayoría de estas son comerciales. En esta tesis se utilizó una implementación de RTI llamada poRTIco (Pokorny y otros, 2008)que tiene certificación de la IEEE y es de uso libre.

La figura 6.19 muestra la estructura del federado de acuerdo a la definición dada. El federado incluye un módulo denominado embajador federado (FederateAmbassadorE2M) que implementa en una forma particular los servicios que el federado debe prestar al RTI durante su interacción, esto es, las funciones *call back*. Además incorpora un módulo denominado embajador RTI (RTIAmbassador) que incluye el conjunto de servicios que el federado puede solicitar al RTI, este último es provisto en el framework *poRTIco*.

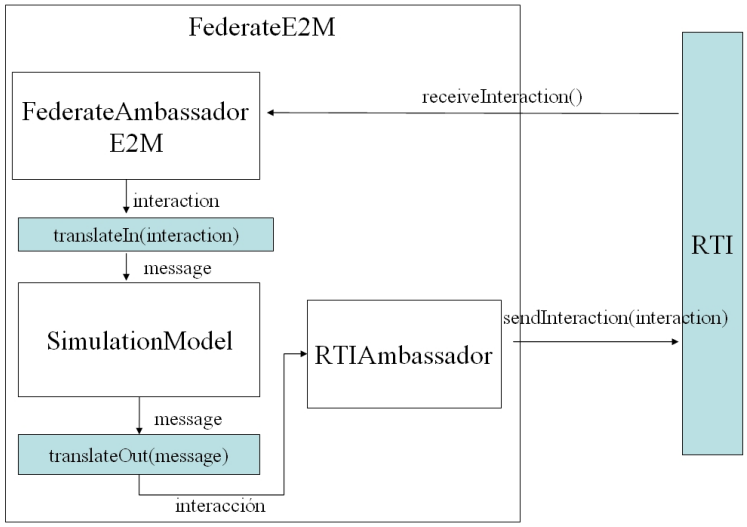


Figura 6.19: Estructura del Federado

El framework *poRTIco* utilizado, provee una clase abstracta llamada *FederateAmbas-*

sador que define la interfaz que un embajador federado debe cumplimentar de acuerdo a la especificación. En la implementación que se está describiendo se especializó *FederateAmbassador* en la clase concreta *FederateAmbassadorE2M* que implementa los métodos especificados en la superclase. El *FederateAmbassadorE2M* debe interactuar con el modelo de simulación con el objetivo de informar sobre las interacciones recibidas desde otros federados, los permisos para avanzar el tiempo de simulación, la actualización de los atributos, los puntos de sincronización con otros federados, etc.

El módulo *RTIAmbassador* y el *RTI*, se tomaron directamente del framework *poRTIco*. El módulo de simulación es una instancia de *SimulationModel*. Los módulos *translateIn(interaction)* y *translateOut(message)* representan la transformación de las interacciones en mensajes (*translateIn(interaction)*) que puedan ser interpretados por los modelos DEVS que forman el modelo de simulación y la transformación de los eventos de salida que aparecen por los puertos de salida del modelo de simulación DEVS en interacciones que puedan ser interpretadas por otros federados (*translateOut(message)*).

El diagrama de clases de la figura 6.20 sintetiza la definición de la clase *FederateE2M*.

La clase FederateE2M tiene como componente una instancia de *SimulationModel* e implementa la interfaz *IODevs*. Esto quiere decir que *FederateE2M* es un federado HLA pero también es un modelo DEVS. Luego en un entorno distribuido, *FederateE2M* representa el modelo DEVS de mayor nivel, el cual puede ejecutar bajo HLA gracias al coordinador asociado que dirige la simulación: *CoordinatorE2M* (cuya lógica fue presentada anteriormente).

CoordinatorE2M es una subclase de *coordinator* provista por el framework *DEVS-JAVA*, que redefine el método *simulate*, incorporando las modificaciones necesarias en el ciclo de simulación DEVS de manera de transformar el esquema optimista en el manejo del tiempo en un esquema conservativo para lograr de esta forma ejecutar un modelo DEVS bajo HLA.

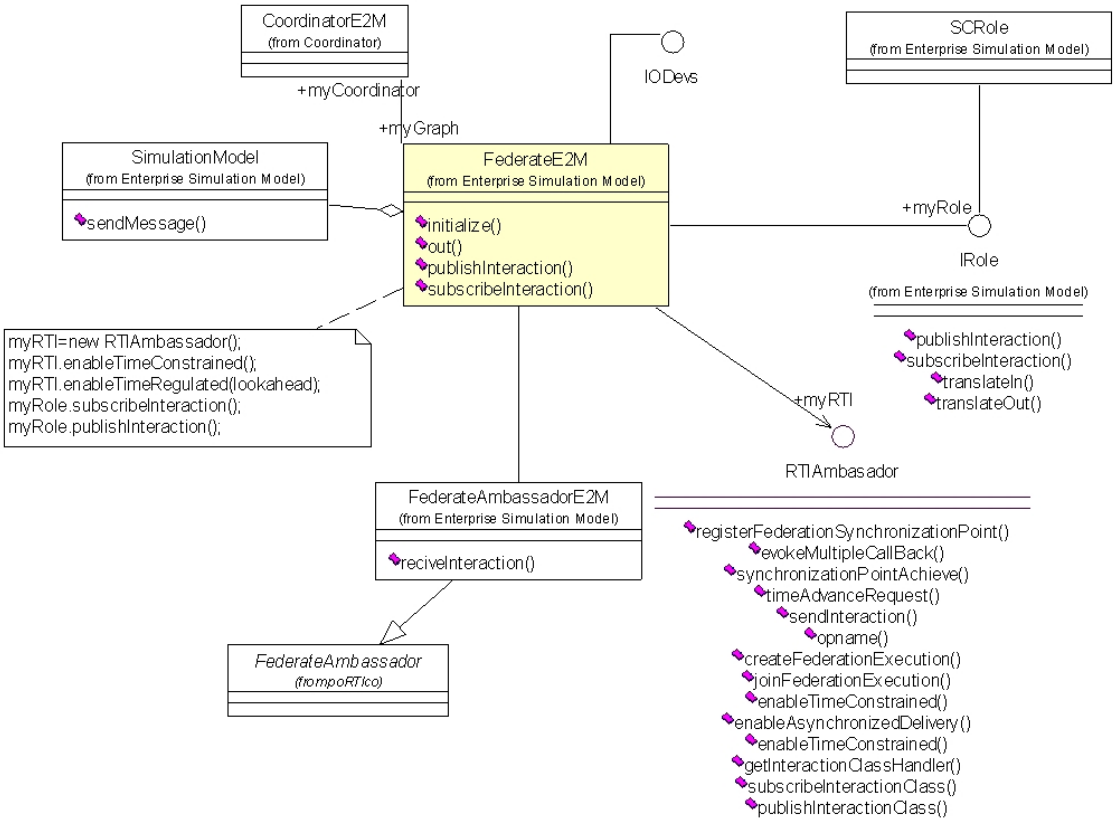


Figura 6.20: Diagrama de cases para FederateE2M

La clase *FederateAmbassadorE2M*, como se dijo anteriormente, implementa en una forma particular las funciones call back definidas en la especificación de la interfaz de HLA. Por ejemplo, la figura 6.21 muestra el código que implementa el método *receiveInteraction()* en la clase *FederateAmbassadorE2M*:

Se puede observar en el código que el embajador federado delega en el federado asociado (*myFederate*) el tratamiento de la interacción que se recibió. El diagrama de interacción de la figura 6.22 muestra la secuencia de mensajes que se genera cuando el federado recibe una interacción.

Así el federado delega en su rol asociado el tratamiento de la interacción recibida. El rol es el responsable de traducir la interacción y crear el mensaje DEVS (instancia

```
public void receiveInteraction(InteractionClassHandle arg0,
                               ParameterHandleValueMap arg1, byte[] arg2, OrderType arg3,
                               TransportationType arg4, LogicalTime arg5, OrderType arg6,
                               RegionHandleSet arg7) throws InteractionClassNotRecognized,
                               InteractionParameterNotRecognized,
                               InteractionClassNotSubscribed,
                               FederateInternalError {
                    try{
                               myFederate.receiveInteraction(arg0, arg1, arg5);
                    }}
```

Figura 6.21: Código del método receiveInteraction

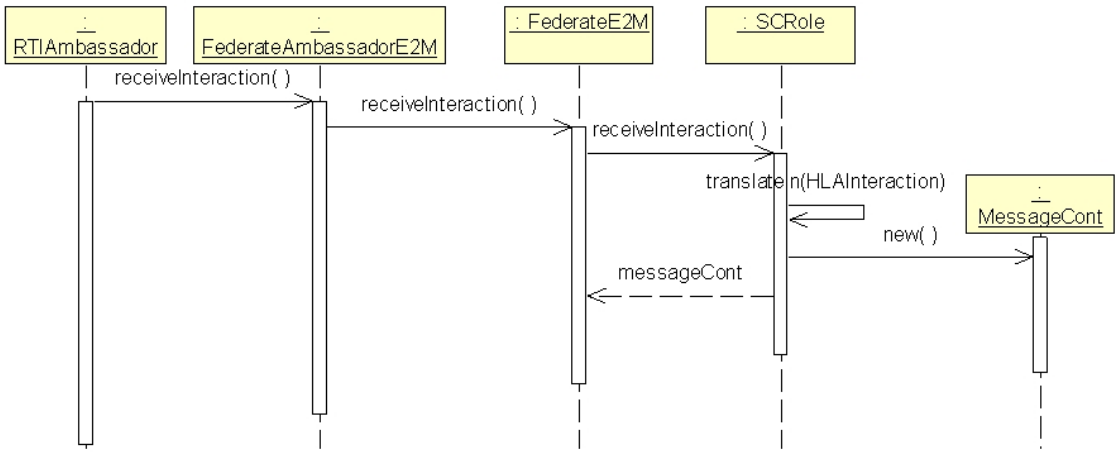


Figura 6.22: Diagrama de secuencia correspondiente a recibir una interacción

de *MessageCont*) el cual será inyectado en el puerto de entrada del federado para su tratamiento posterior.

La clase *SCRole* implementa la interface *IRole*, siendo responsable del tratamiento de las interacciones entre los federados. Esta interface define los métodos *publishInteraction, subscribeInteraction, translateIn* y *translateOut*. Esta clase fue incorporada con el objetivo de lograr la composición de los simuladores. El problema que se plantea al usar simulación distribuida es que el solo hecho de adecuarse al estándar HLA no es suficiente para facilitar la composición (David y Anderson, 2003). Esta adecuación solo proporciona interoperabilidad a nivel sintáctico, dependiendo la composición fuertemente de

la definición del FOM. La composición es definida como "*la capacidad de seleccionar y ensamblar componentes en un entorno de simulación válido y completo, para satisfacer requerimientos específicos de usuario*" (DoD, 1995) (Petty y Weisel, 2003).

Si bien la definición del rol propuesto es un paso importante en lograr la composición en simuladores que representan el comportamiento de procesos de empresa como los definidos en esta tesis, no alcanza para lograrla completamente, sin embargo representa el inicio de un largo camino a seguir donde intervienen otros aspectos (como ser el uso de metadatos, ontologías, utilización de FOM modulares y el uso de tecnología tales como estándares) que están fuera del alcance de esta tesis.

La clase *SCRole* tiene tres subclases: *Make, Deliver* y *Source*, implementando de diferentes formas los métodos de su superclase. De esta forma, cada rol conoce cuales interacciones debe publicar y cuales subscribir, como también conoce la forma de traducir las interacciones en mensajes DEVS y vice versa.

Por ejemplo, el código que se presenta en la figura 6.23 muestra parte de la implementación del método *publishInteraction* (figura 6.23 (a)) y *subscribeInteraction* (figura 6.23 (b)) implementado en la clase *Make*.

Este código muestra las acciones necesarias para publicar la interacción *DeliverGoods* y subscribir la interacción *AskGoods* con sus parámetros correspondientes. Estas interacciones son almacenadas en una tabla (*tablaInteraciones*) para poder interpretar las interacciones que llegan desde el RTI y que tendrán que ser transformadas en mensajes.

Luego, el código necesario en la clase *FederateE2M* para publicar o subscribir interacciones se simplifica a delegar en su rol asociado esta responsabilidad como se muestra en el código siguiente:

```
public void publicInteraction()
    {
        role.publicInteraction(myRTI);
    }
```

```
public void
publishInteraction(RTIambassador myRTI)
{           // federado MAKE //
            InteractionClassHandle hi;
            ParameterHandle pi;
            try {
  ////////////////////////////////////////////
 // interaccion DeliverGoods //
 ////////////////////////////////////////////
hi =
myRTI.getInteractionClassHandle("DeliverGoods");
pi = myRTI.getParameterHandle(hi,"itemID");
tablaparametros.put(pi,hi);
pi = myRTI.getParameterHandle(hi, "date");
tablaparametros.put(pi,hi);
pi = myRTI.getParameterHandle(hi,"quantity")
tablaparametros.put(pi,hi);
tablaInteracciones.put(hi,"DeliverGoods");
myRTI.publishInteractionClass(hi);}
```

(a)

```
public void
subscribeInteraction(RTIambassador myRTI)
InteractionClassHandle hi;
ParameterHandle pi;
try {
     ////////////////////////////////////
    // interaccion AskGoods //
   ////////////////////////////////////
hi =myRTI.getInteractionClassHandle("AskGoods);
pi = myRTI.getParameterHandle(hi,"item");
tablaparametros.put(pi,hi);
pi = myRTI.getParameterHandle(hi,"customer
    orderNro");
tablaparametros.put(pi,hi);
pi = myRTI.getParameterHandle(hi, "quantity");
tablaparametros.put(pi,hi);
tablaInteracciones.put(hi,"AskGoods");
myRTI.subscribeInteractionClass(hi);}
```

(b)

Figura 6.23: Código para publicar y subscribir interacciones

De esta forma, un federado puede cambiar el conjunto de interacciones publicadas e interacciones subscriptas cambiando su rol. Así también, la traducción de mensajes DEVS a interacciones y vice versa queda como responsabilidad del rol, de manera que no es necesario re-programar el federado para reflejar estos cambios. El código que se muestra en la figura 6.24 corresponde a la implementación del método que transforma una interacción en un mensaje DEVS implementado en la clase *Make*. La figura 6.24 (a) muestra el código del mensaje *receiveInteraction* mientras

Claramente se puede deducir que el rol debe conoce la interface definida para el modelo de simulación en el momento de realizar las traducciones, pudiendo interpretar la interacción que arriba y transformarla en un evento inyectado al modelo DEVS de simulación.

```
public MenssageCont receiveInteraction(InteractionClassHandle handler, ParameterHandleValueMap arg1,
                                       LogicalTime tiempo, FederateE2M federadoE2M)
{
          String valorInteraccion = tablaInteracciones.get(handler).toString();
        // transformo la interacción en mensaje DEVS y lo agrego al mensaje de salida //
        Collection parametros = arg1.values();
        return translateIn(valorInteraccion, arg1, federadoE2M);
}
```

(a)

```
public MenssageCont translateIn(String unaInteraccion,  ParameterHandleValueMap parametros,
                                FederateE2M federadoE2M)
{                   Vector lista = new Vector();
                    lista.add(federadoE2M); //generador del mensaje
                    Set s = parametros.keySet();
                    Iterator i = s.iterator();
                    while(i.hasNext()){
                              byte[] valor = parametros.get(i.next());
                              Integer valorEntero = new Integer(valor.toString());
                    // lista tiene el modelo que genera el mensaje y luego el conj. de parámetros
                              lista.add(valorEntero.intValue());
                    }
                    MenssageCont con= new MenssageCont("generate",lista);
                    return con;
}
```

(b)

Figura 6.24: Código correspondiente a transformar una interacción en mensaje DEVS

6.3. Conclusiones

En este capítulo se presentaron las reglas de transformación del modelo conceptual de la empresa en un modelo de simulación equivalente. Se presentaron las características de diseño de la capa de simulación y se explicó en detalle el proceso de transformación del modelo de empresa en el modelo de simulación. Finalmente se presentaron los mecanismos de ejecución que la arquitectura provee. Para la simulación en entorno distribuido se presentó en detalle la creación del federado, la máquina de simulación que es necesaria para dirigir una simulación de modelos DEVS en entornos distribuidos y el uso de roles como mecanismo para mejorar la interoperabilidad entre los federados que participan de una federación.

7

Caso de Estudio: Simulación Local y Distribuida en el Contexto de una Fábrica de Yerba Mate

En este capítulo se presenta un ejemplo del uso de la arquitectura para el diseño y análisis de una cadena de suministro. Se describe una empresa de producción de yerba mate con sus procesos, recursos y los procesos externos con los otros miembros de la cadena. El modelo de empresa es construido y simulado localmente para luego ser parte de una federación que se construye con el fin de analizar los procesos de la cadena. Finalmente los resultados son evaluados.

La cadena de suministro está compuesta por una fábrica de yerba mate localizada en Misiones (Argentina) y un centro de distribución localizado en nuestra zona (Santa Fe - Argentina). El objetivo principal es mostrar el uso de la arquitectura en el contexto de cadenas de suministro. En el desarrollo del ejemplo se utiliza un modelo de objeto para la federación (FOM) que fue definido específicamente para este caso, sin embargo se continúa trabajando para mejorar la posibilidad de utilizar diferentes FOM de acuerdo a las necesidades de los participantes y de la estructura de la empresa.

El capítulo está organizado de la siguiente manera: primero se presenta la descripción de los procesos internos de la fábrica de yerba mate y posteriormente se desarrolla el modelo de empresa para dicha fábrica y se ejecuta la simulación local mostrando los

resultados obtenidos. A partir de estos resultados, se proponen mejoras en los procesos, los cuales son evaluados usando nuevamente simulación local. A continuación se realiza el análisis de las relaciones con otros miembros de la cadena. La característica distribuida de la cadena se debe a la presencia de un distribuidor del producto yerba mate localizado en la ciudad de Santa Fe. Luego se define el modelo de objetos de la federación, FOM que será utilizado en el desarrollo de este ejemplo que representa las relaciones colaborativas entre las dos empresas que integran la cadena de suministro. Finalmente se ejecuta la simulación distribuida y se presentan los resultados obtenidos.

7.1. Fábrica de Yerba Mate

La fábrica que se analiza, es una cooperativa de más de 100 pequeños productores de yerba mate. Esta planta comenzó siendo un molino secadero de hojas verdes de yerba que le permitió a sus socios comercializar el producto agregándole valor. Más tarde se comenzó a comercializar yerba a granel a las envasadoras. La yerba a granel, es un producto que se utiliza para fabricar la yerba mate que se consume en los hogares y consiste en bolsas de 50 kg. de yerba canchada estacionada por dos años. Luego, se agregó al molino una planta envasadora con el fin de producir y comercializar yerba mate con marca propia.

La yerba mate es un producto que se cosecha de abril hasta septiembre. Durante este período el molino trabaja recibiendo 11.400 toneladas de producto en bruto (hojas verdes de la planta). El procesamiento de la materia prima, para obtener el producto final es un proceso que demora 2 años, desde que se reciben las hojas del producto bruto hasta que el mismo pueda ser envasado y comercializado.

Para organizar la presentación del ejemplo, se seguirán los pasos propuestos en el capítulo tres para el uso de esta arquitectura. Primeramente se desarrolla el modelo de

empresa o modelo conceptual de la fábrica utilizando el lenguaje Coordinates (EM). En la realización de esta etapa, se presentará la vista de tareas y luego la vista dinámica y vista del dominio. Una vez construido el EM, se procede a ejecutar la simulación local de los procesos. Luego, se selecciona la ejecución de la simulación distribuida eligiendo como rol para esta fábrica el rol MAKE (presentado en el punto 6.2.1). Finalmente se presentan los resultados obtenidos.

Comenzando con el diseño del EM, es posible identificar tres procesos troncales: proceso de producción de yerba canchada, proceso de envasado o empaquetado y proceso de comercialización. Estos procesos se representan utilizando Coordinates para lo cual es necesario identificar las tareas y los recursos que intervienen en los mismos. Las siguiente secciones describen cada uno de estos procesos. En el proceso de construcción de estas vistas, no es necesario seguir un orden estricto, sin embargo se sugiere que antes de crear la vista dinámica, estén definidas las tareas que intervienen en los diagramas de estado de los recursos no siendo esto una restricción ya que pueden crearse las tareas a medida que se las necesite utilizar en el diagrama.

7.1.1. Vista de Tareas

Proceso de producción. El proceso de producción comienza cuando los camiones de los distintos productores llegan a la fábrica depositando las ramas de hojas verdes de yerba mate en el playón del secadero. Aproximadamente se reciben 20 toneladas de hojas verdes por día. Un empleado, utilizando un rastrillo mecánico, junta las ramas del playón del secadero y la deposita en una cinta transportadora con un caudal de 8.000 kg/h. La cinta transportadora lleva las hojas verdes hacia un molino circular donde se realiza el *sapecado*. Allí se expone a la hoja a fuego directo durante 1 minuto. Tiempo en que la hoja se vuelve crocante y pierde el 20 % de su peso. Este proceso es necesario

que se realize antes de las 24hs. de cortada la hoja de la planta. Es por eso que el secadero trabaja intensamente durante el período de cosecha. Más tarde, la hoja pasa a un horno de entre 80 y 100°C por 2 min. donde la hoja sigue perdiendo humedad, pero sin llegar a tostarse. Este proceso es el *secado* de la hoja. Los hornos que se utilizan en el secado de la hoja, son controlados manualmente por un empleado. Si el horno pierde temperatura, la hoja no llega a secarse bien perdiendo calidad el producto final. Es por ello que el empleado debe estar atento y descartar aquellas hojas que no están bien secas a la vez que debe aumentar la temperatura del horno agregando leña.

De allí pasa a una cinta transportadora donde la yerba es sometida durante 4 hs. a corrientes de aire caliente. Seguidamente, se produce el *canchado*, donde se tritura la hoja en pedazos pequeños de no más de 1 cm. y se la separa del palo. La yerba canchada se fracciona en bolsas de arpillera de 50kg la que es estacionada durante dos años en un deposito acondicionado.

Se pueden identificar en este proceso, cuatro etapas: secar, canchar, transportar y almacenar. La tarea de secado involucra las siguientes tareas: recepción de la hoja verde, sapecado de las hojas y deshidratado de las hojas. Los recursos que se utilizan en estas distintas tareas son: empleados, rastrillo mecánico, hornos, cintas transportadoras, yerba verde, yerba sapecada, camión y yerba canchada. Representando estas tareas se obtiene la versión del proceso como se muestra en la figura 7.1, el cual fue construido con la herramienta Coordinates Workbench.

Las tareas que intervienen en la descripción del proceso no aparecen vinculadas por relaciones temporales debido a que la dependencia entre estas está dada por la existencia de recursos para procesar. Así por ejemplo, se puede interpretar que mientras exista *yerba sapecada*, la misma va a ser consumida por la tarea *canchar*. Una relación temporal *antes-que* por ejemplo, estaría indicando que *canchar* solo es posible ejecutarla si la tarea anterior terminó su ejecución, lo cual no sería correcto dado que las

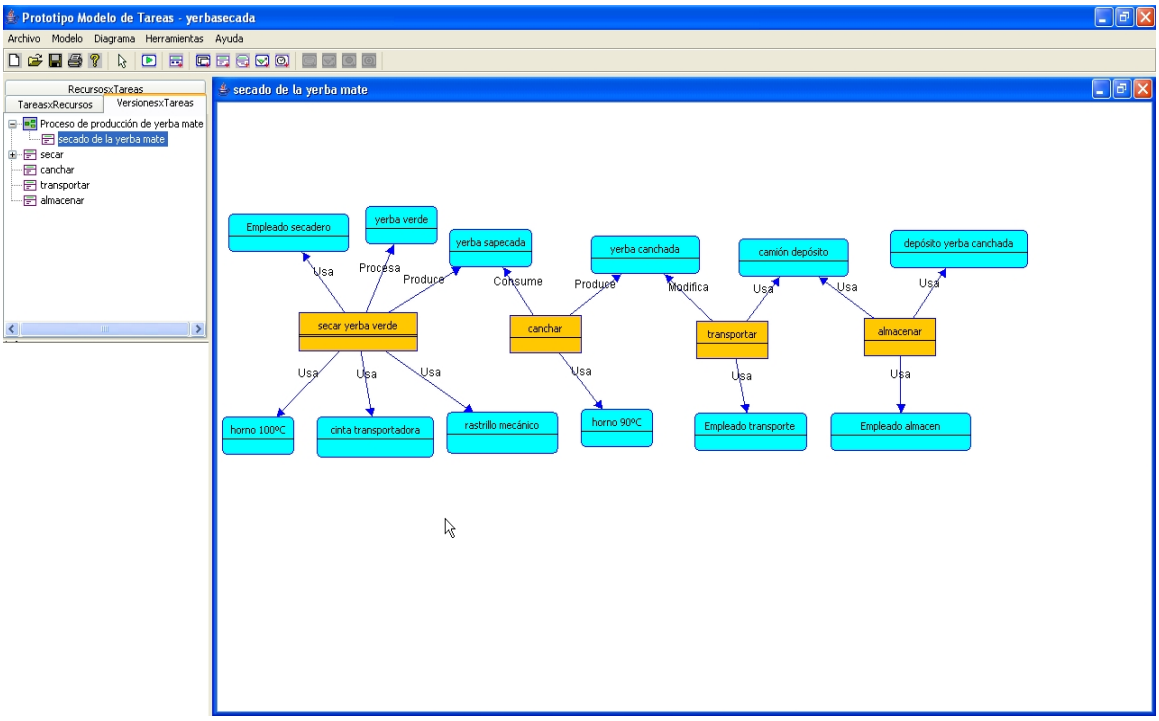


Figura 7.1: Ventana de Coordinates Workbench: Proceso de producción de yerba mate

tareas se pueden estar ejecutando en paralelo mientras existan recursos que puedan ser consumidos por las mismas.

Siguiendo con el ejemplo, la tarea *secar (secar yerba mate)*, se la representa como una tarea compuesta (doble línea debajo del nombre de la tarea en el gráfico) descripta a través de una versión de tareas que muestra su descomposición en las subtareas *recepcion, sapecar* y *deshidratar* como muestra la figura 7.2.

Los recursos que están asociados a la tarea *secar* en la figura 7.1 están también presentes en su versión de tarea asociada. De esta manera se muestra cómo la tarea modifica a los recursos.

Proceso de envasado. Recién cuando se cumplen los dos años de estacionamiento para la yerba canchada, la misma es llevada a la planta de envasado donde se la fracciona en paquetes de 1kg y de 1/2kg para su comercialización.

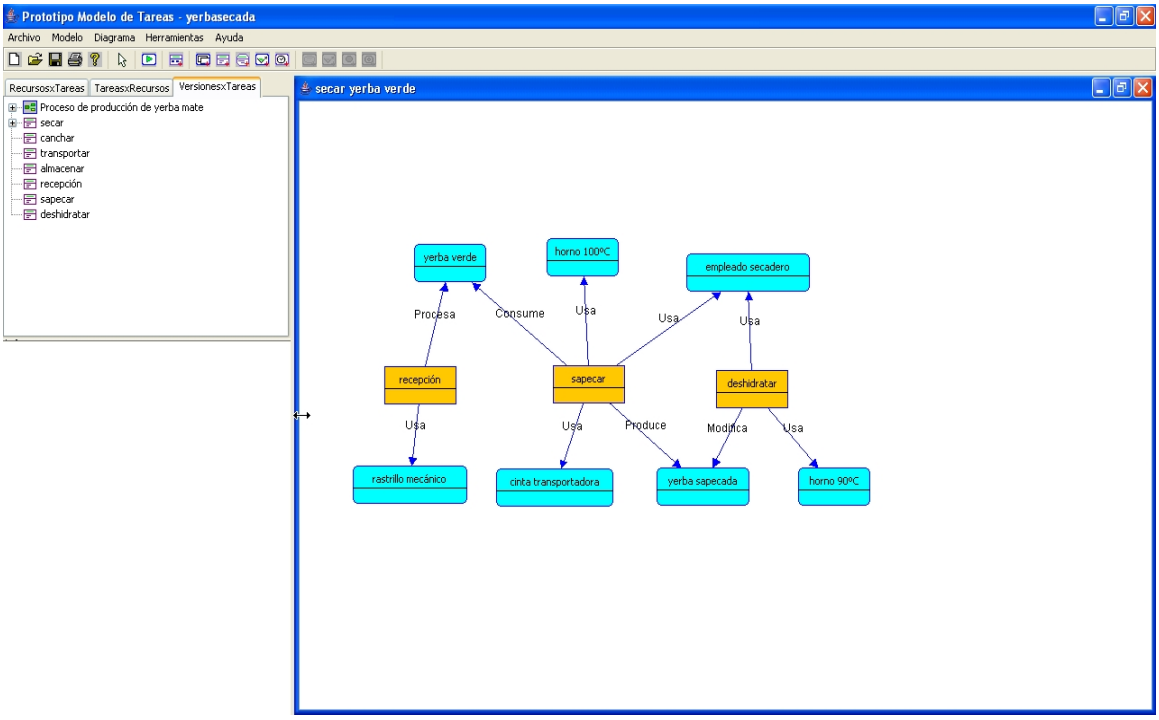


Figura 7.2: Ventana de Coordinate Workbench: Versión de tarea de Secar

Las bolsas de 50 kg. de yerba canchada estacionada, son transportadas desde el depósito a la planta de envasado en camiones. Las bolsas son vaciadas en una zaranda para eliminar los cuerpos extraños, luego es triturada, y en función de la calidad, agrega o no palos. Existen tres tipos de calidad de yerba: (i) yerba común con un 35 % de palos, (ii) yerba especial con un 20 % de palos, (iii) yerba selección con un 14 % de palos y (iv) yerba selección especial sin palos (el palo es el responsable de la acidez que puede provocar el mate). Una vez triturada la yerba pasa a un embudo que regula el paso de la yerba mate, esta máquina es programada de acuerdo al paquete que se esté llenando. Un empleado controla el peso de los paquete tomando una muestra de aproximadamente 1 paquete de cada 20 y lo pesa. Luego, pasa por una máquina que cierra el paquete y pega una estampilla correspondiente al impuesto. Finalmente los paquetes son agrupados de a 20 (los de 1kg) o 30 (los de 1/2 kg). Este pack es transportado a depósito para su venta

posterior. La figura 7.3 muestra el proceso de envasado. Como se observa existen dos tareas que tienen versiones asociadas. Estas tareas corresponden a *triturar* y *empaquetar yerba*.

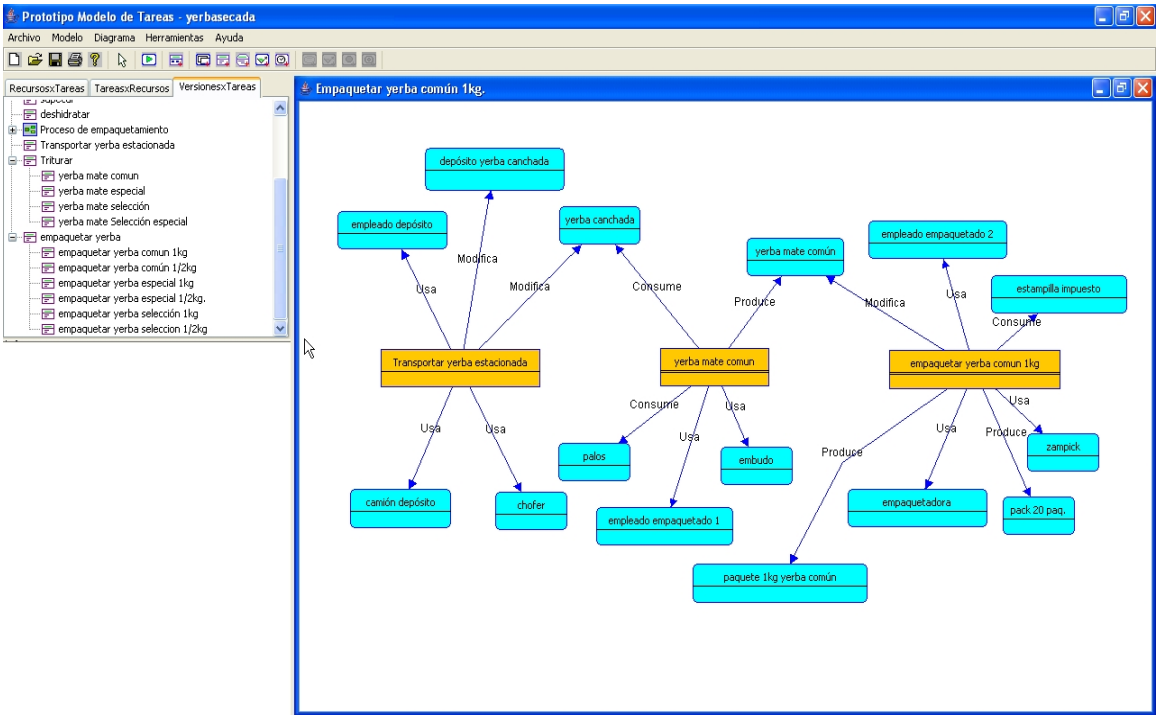


Figura 7.3: Ventana de Coordinates Workbench: Proceso de envasado de la yerba mate

La primera, se encarga de agregar los palos, teniendo una versión por cada calidad de yerba que existe, en el gráfico 7.3 esta tarea aparece con el nombre de una de sus versiones asociadas llamada *yerba mate común*, la cual es la versión default con la que participa en la versión que se describe. La segunda tarea, *empaquetar yerba, (empaquetar yerba común 1kg.)*, tiene una versión por cada tipo de paquete, esto da una combinación entre las calidades de la yerba y el tamaño del paquete. No se muestran todas estas versiones por razones de simplicidad, pero las mismas pueden observarse formando parte de la estructura de descomposición que se muestra sobre la izquierda de la ventana del workbench en la figura 7.3, la cual representa la vista del repositorio.

Proceso de comercialización. El proceso de comercialización se realiza a través de distribuidores. Existen dos tipos de ventas: las ventas directas de fábrica y las ventas a través del distribuidor. La primera, corresponde a la venta de grandes volúmenes de mercadería por lo cual, el distribuidor toma el pedido y el mismo es cargado directamente en fábrica y descargado en el cliente. La segunda, se trata de volúmenes que pueden ser satisfecho con el stock que tiene el distribuidor. El distribuidor envía órdenes de entrega a fábrica cuando el stock está por debajo de un mínimo establecido. Es importante destacar que todas aquellas órdenes que no son satisfechas en tiempo y forma, es venta perdida dado que, se trata de un producto de consumo masivo el cual puede ser sustituido. La característica particular de este tipo de producto es que el mismo es de alta rotación, altos volúmenes de venta y de bajo costo (Ballou, 1999). Es un producto sustituible, es decir que en caso de no encontrarse ese producto, el consumidor lo reemplazará por otro. Esto influye en la comercialización: es importante que el producto sea encontrado en las góndolas del supermercado. Al ser un producto de bajo costo, su costo de distribución es alto, por lo cual es necesario aumentar el volumen para poder justificar el costo. El nivel de satisfacción del cliente, expresado en términos de disponibilidad y accesibilidad del producto debe ser elevado para asegurar un alto grado de identificación del cliente con el producto.

Las órdenes de entrega provenientes de los distribuidores son procesadas diariamente y cargadas en camiones. Al llegar al depósito del distribuidor, se firma en conformidad el remito. Al finalizar la semana el distribuidor realiza pagos a fábrica por la mercadería que fue cobrada y entregada a los clientes. Cada 6 meses se hace un control de stock en depósito del distribuidor.

La figura 7.4 muestra el proceso de comercialización para el caso de venta directa, desde que la fábrica recibe la orden del cliente hasta que la misma sea entregada al cliente final. En este caso, la fábrica carga el camión, imprime la factura y entrega

la mercadería. El proceso está compuesto por las tareas: *procesar orden del cliente, preparar orden del cliente, cargar camión* y *entregar mercadería*.

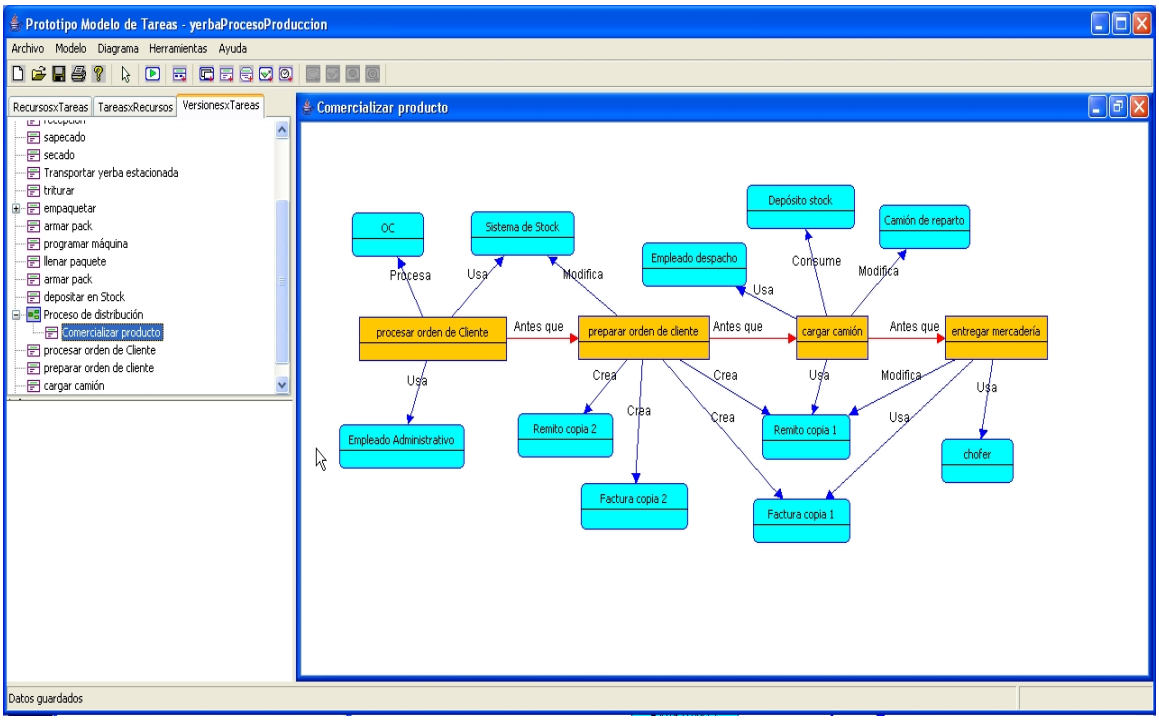


Figura 7.4: Ventana de Coordinates Workbench: Proceso de comercialización de la yerba mate

En el caso que la orden que se recibe corresponde a la reposición de mercadería en el distribuidor, la fábrica recibe la orden de entrega, la procesa pero en este caso sólo imprime el remito dado que el distribuidor vende por cuenta y orden de fábrica. La facturación se realiza en un procedimiento independiente a través del sistema de facturación de acuerdo al remito que genera el distribuidor. Estos procesos están representados en el modelo del distribuidor y no corresponde representarse en el modelo de la fábrica. La figura 7.5 muestra el proceso de reposición de mercadería en el distribuidor.

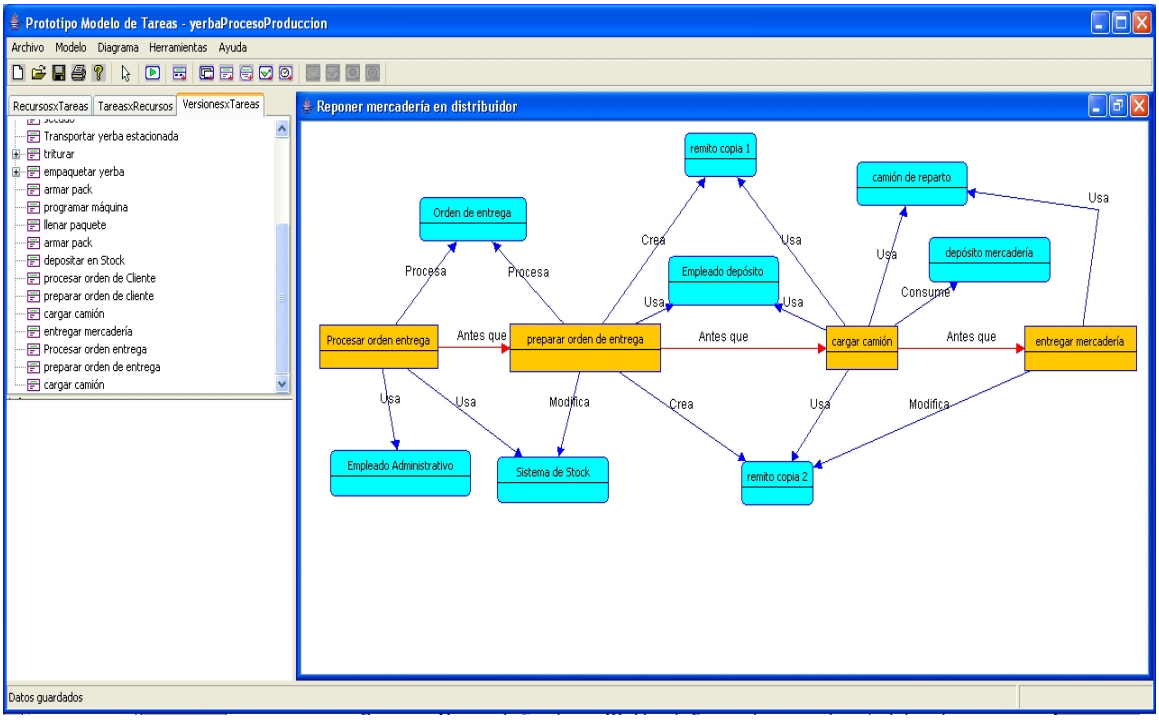


Figura 7.5: Ventana del workbench: diagrama de tareas correspondiente a reponer mercadería en el distribuidor

7.1.2. Vista Dinámica.

En esta vista se intenta mostrar desde la perspectiva de los recursos, cómo estos se ven afectados al participar de las tareas, lo cual se representa por medio de diagramas de transición de estados (DTE) de los mismos. Un ejemplo de un DTE asociado a un recurso se muestra en la figura 7.6, en donde se modela el recurso *Empleado*, el cual tiene un ciclo de vida típico de los recursos que son usados por las tareas.

Por lo general estos recursos pasan a estar ocupados mientras participan de la tarea, e imposibilitados de participar de otras tareas y luego quedan libres. Un diagrama similar describirá a otros recursos, como por ejemplo: camión, cinta transportadora, rastrillo mecánico, hornos, etc.

La figura 7.7 muestra el ciclo de vida del recurso *yerba canchada*.

Este producto, es producido en el molino como consecuencia de la ejecución de

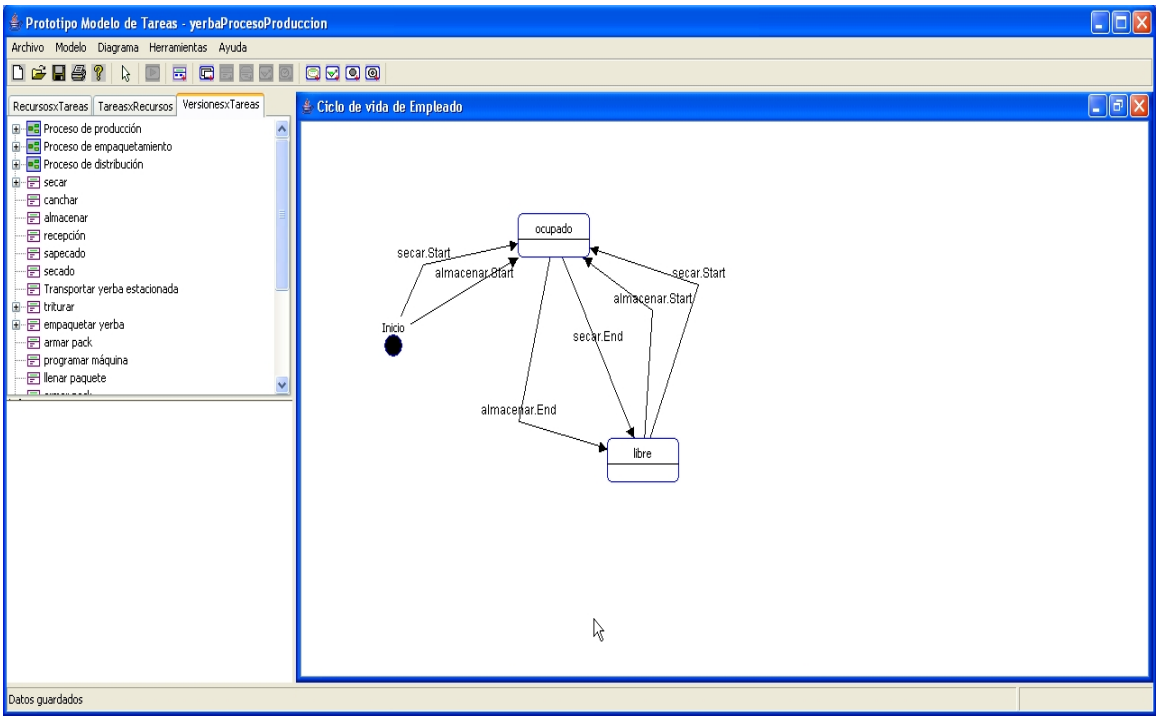


Figura 7.6: Ventana de Coordinates Workbench: diagrama de transición de estado del recurso Empleado

la tarea *canchar*, actividad que consume la *yerba sapecada* y produce yerba de otro tipo, llamada *canchada*. El producto, luego de ser elaborado, es depositado en galpones acondicionados. La tarea *almacenar*, es la responsable del cambio de estado de este recurso: cuando está en el sector de producción, pasa al sector depósito y permanece allí por dos años. La tarea *transportar yerba estacionada* deja al recurso en el estado *en sector envasado* al finalizarse la misma. Finalmente, la tarea *triturar* consume el recurso *yerba canchada* llevándola a su estado final. Esto se define de esta manera dado que la tarea *triturar* genera un nuevo recurso, *yerba mate*, a partir del recurso *yerba canchada* al agregarle los palos y triturarla. Con este proceso de *triturar*, la *yerba canchada* como tal deja de existir.

Sería posible continuar mostrando los otros diagramas de transición de estados pero no se considera apropiado para el ejemplo que se quiere mostrar ya que se estarían

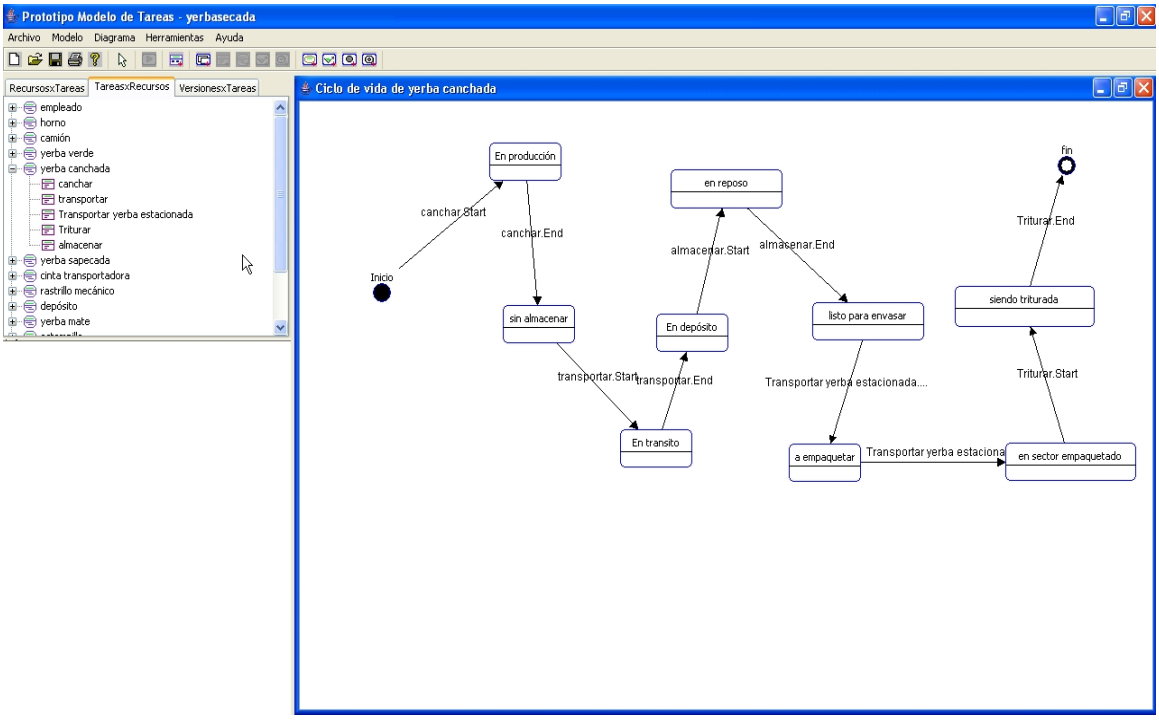


Figura 7.7: Ventana del Workbench: DTE del recurso Yerba Canchada

aportando detalles irrelevantes que más que aclarar pueden distraer la atención del lector.

7.1.3. Vista del Dominio

Esta vista representa las relaciones de los recursos entre sí. El diagrama de la figura 7.8 muestra las relaciones definidas entre algunos de los recursos participantes de la vista tarea. Así se observa que el recurso *Documento Comercial* se clasificó en: *Remito, Factura* y *Orden Cliente*. Luego, los recurso *Remito* y *Factura* se descomponen de acuerdo a sus copias identificando cada copia de estos documentos comerciales como componentes del mismo.

Se identificaron también los distintos tipos de yerba que intervienen en los procesos: *yerba canchada, yerba sapecada* y *yerba verde* como subclases de la clase *Yerba*. El re-

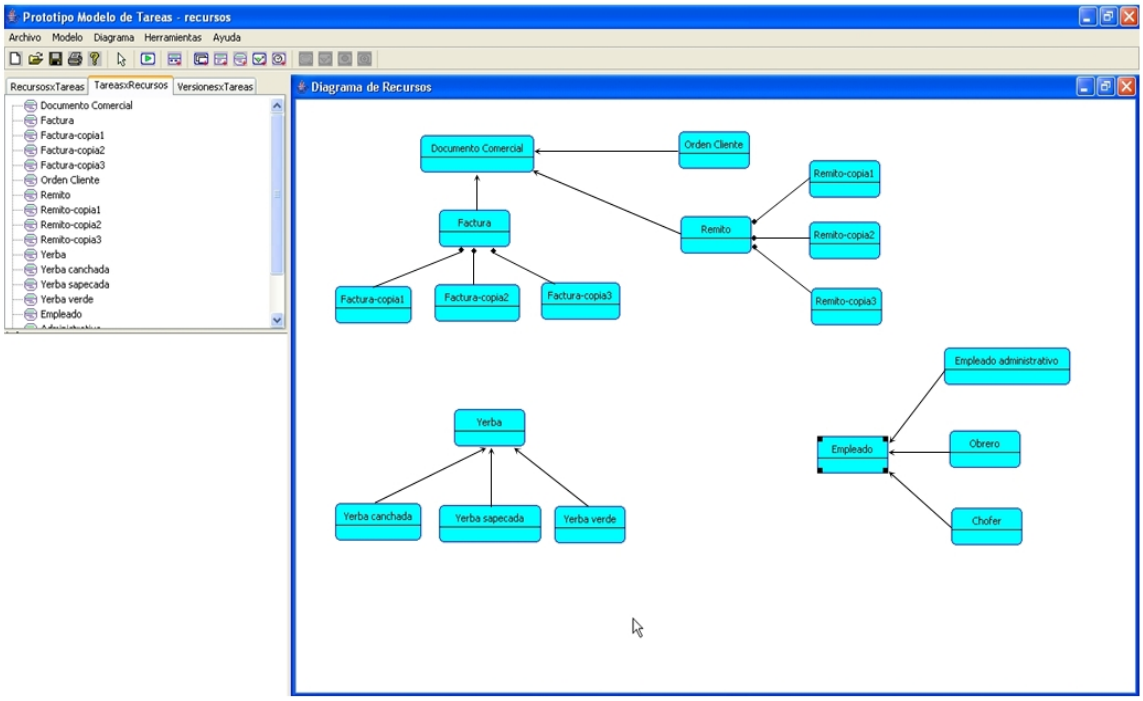


Figura 7.8: Ventana del Workbench: Vista Dominio

curso *Empleado* aparece subclasificado en: *Empleado Administrativo*, *obrero* y *chofer*. Si bien, no aparecen en este diagrama la totalidad de los recursos, estas mismas relaciones pueden establecerse para identificar las distintas máquinas herramientas que se utilizan en los procesos, los sistemas de información y los productos elaborados como los pack de paquetes de yerba envasada en distintos tamaños.

Esta vista tiene como finalidad la representación de las relaciones entre recursos, sin embargo, no es necesario que todos los recursos estén participando de diagramas de este tipo. Aún cuando un recurso no aparezca en el diagrama formará parte de la vista del dominio cuando sea definido en otros diagramas, o en la vista del repositorio.

7.2. Simulación Local para el EM de la Fábrica de Yerba Mate

Una vez finalizada la actividad de definir las distintas vistas con sus diagramas, se está en condiciones de construir el modelo de simulación y a continuación de ejecutar la simulación local de los procesos presentados.

La figura 7.9 muestra la ventana del workbench donde es posible identificar el tiempo de simulación y ciertas condiciones que pueden identificarse para las tareas y recursos que participan del proceso a simular.

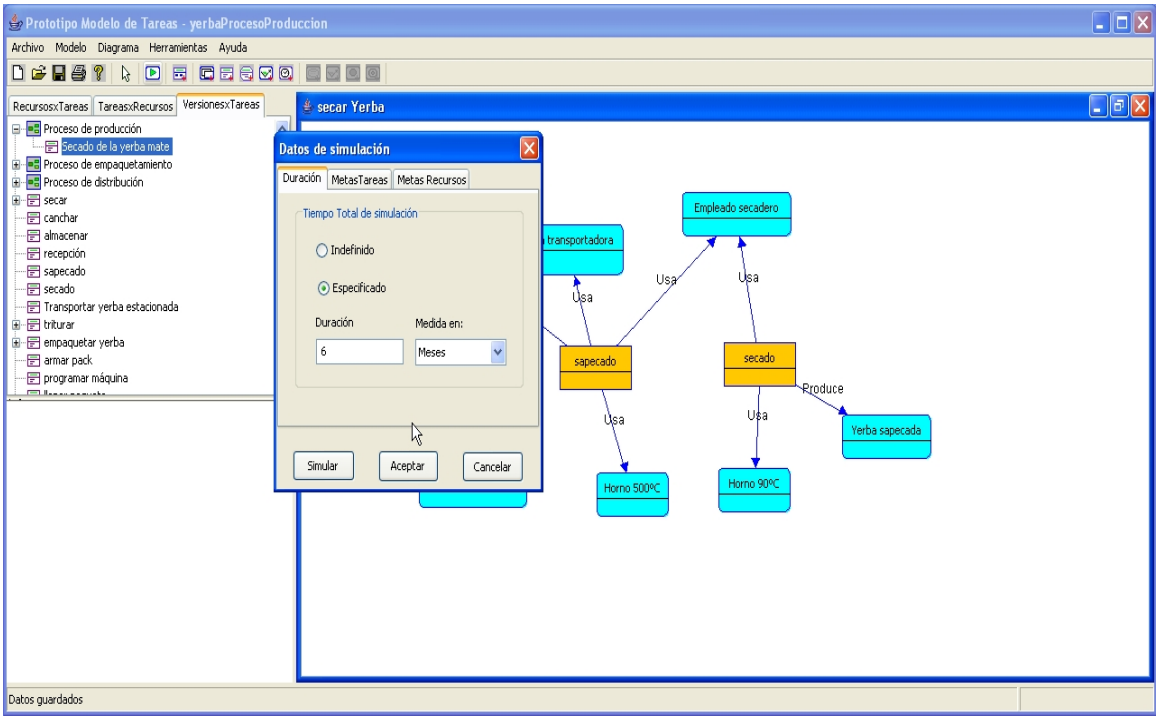


Figura 7.9: Ventana del Workbench: Datos de simulación

Así se especifica como tiempo de simulación 6 meses de manera de poder analizar el proceso completo de producción durante el período de cosecha. El objetivo es poder analizar el proceso de producción para obtener información sobre posibles cuellos de

botellas, pérdida de valor del producto, operaciones ociosas por un tiempo largo, etc. A partir de estos datos el experimento es generado y el *aceptor* se creará para enviar el evento de fin de simulación en el minuto 259.200 correspondiente a la cantidad de minutos que existen en 6 meses considerando meses de 30 día y 24 hs de trabajo por día. Si bien, el período de simulación se indica en meses, internamente el sistema transformará los meses a minutos si esta es la medida más pequeña indicada en las tareas. Si por ejemplo la medida más pequeña fuera días, la traducción se haría en días. El evento que genera el disparo del proceso es la llegada de la hoja verde al depósito de sapecado. Para este ejemplo se indicó que la tarea *recepción* procesa el recurso *yerba verde*. Luego, esta relación está indicando el recurso que será generado por el experimento como eventos que alimentan al modelo de simulación. Al indicarse un período de tiempo en el tiempo de simulación, el objetivo es evaluar el proceso con el fin de determinar la cantidad de *yerba canchada* que se produce durante el período de cosecha. Para el caso de otros procesos, como ser el de comercialización, indicar un período de tiempo en los objetivos de simulación puede ser interpretado como la evaluación del proceso en función de la satisfacción de las órdenes del cliente. En este caso particular se podría analizar cuántas órdenes fueron satisfechas completamente y en tiempo, y cuantas sufrieron demoras. También es posible indicar condiciones de fin de simulación en función de estados de tareas o recursos. En la figura 7.9 puede observarse que existen dos pestañas que tienen la leyenda *metas tareas* y *metas recursos*. Estas pestañas permiten identificar estados de las tareas o estados de los recursos a alcanzar en la simulación. Así es posible identificar otros objetivos en la simulación como:

- estado de recursos: el objetivo se expresa en función de estados de uno o más recursos. Por ejemplo si se quiere evaluar el proceso hasta alcanzar una cantidad de 5000kg de yerba canchada o hasta que las órdenes del cliente alcancen el estado

completa.

- tareas a ser ejecutadas: aquí el objetivo se expresa en función de cuáles tareas deben ser ejecutadas. Por ejemplo se podría determinar evaluar el proceso de simulación hasta que la tarea *entregar mercadería al cliente* se ejecute.

Una vez indicado el objetivo de simulación, el modelo conceptual se traduce en un modelo de simulación el cual es ejecutado. Esta tarea es llevada a cabo en la capa de simulación.

Los gráficos de la figura 7.10 muestran los resultados obtenidos. El gráfico (a) de

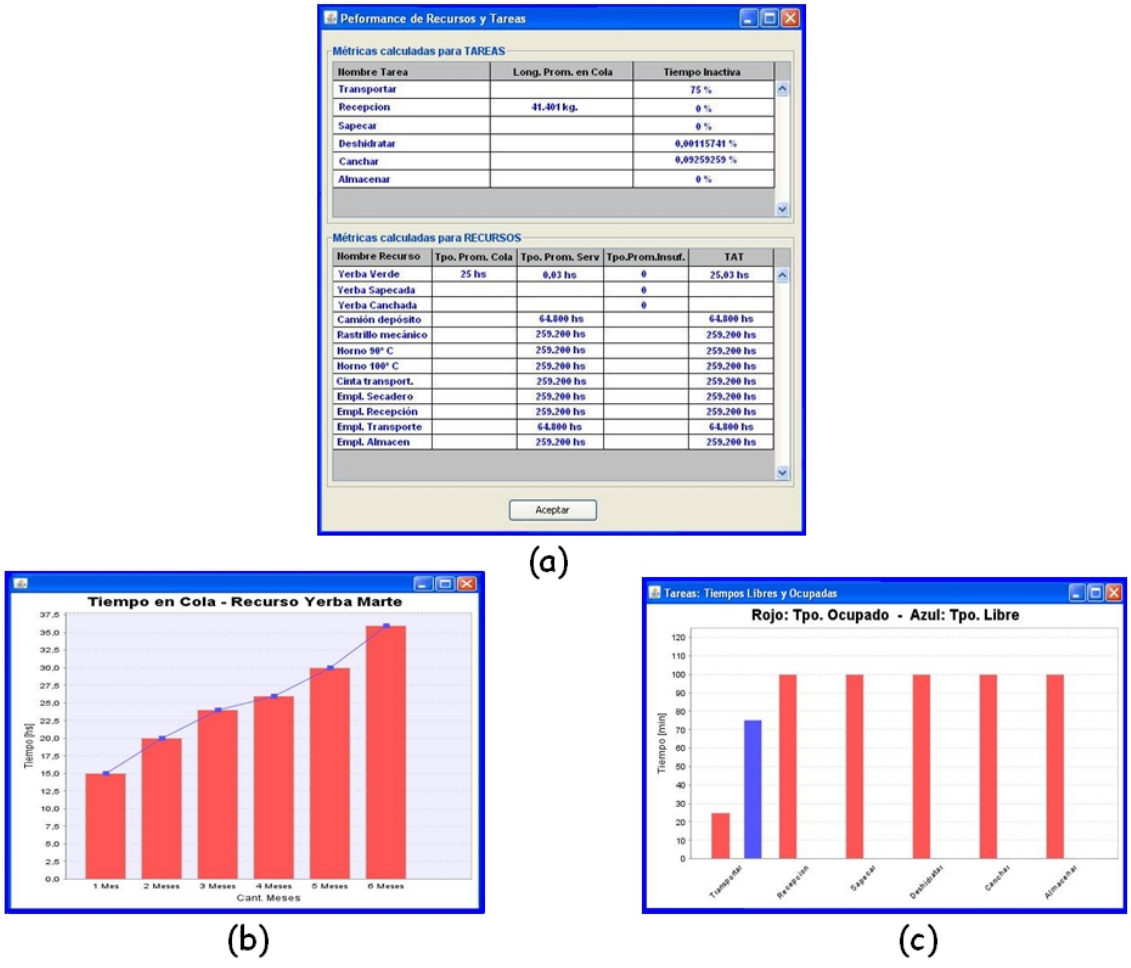


Figura 7.10: Resultados de la simulación Local

la figura 7.10 muestra una tabla con valores numéricos. La parte superior de la tabla representa los valores calculados para las tareas, indicando para cada una la longitud de cola promedio y el tiempo de inactividad. Las tareas que no tienen especificado el valor *longitud de cola* se debe a que las mismas no presentan cola. En este caso, la tarea recepción tiene una longitud de cola promedio de 41401 kg de yerba verde y un tiempo de inactividad de cero. Esto revela una importante falencia en la capacidad de procesamiento. Siguiendo con el análisis, la tabla inferior del gráfico muestra valores de los recursos. Aquí se establece que el tiempo en cola del recurso yerba verde es de 25hs promedio. Este dato está relacionado con el problema anteriormente analizado: la tarea *secar* no tiene la suficiente capacidad como para procesar los recursos que llegan dado que las subtareas de *secar* están trabajando al 100 %. En la descripción del proceso de producción se estableció que las hojas de yerba debían ser secadas antes de las 24hs de cortada de la planta. Esta restricción no se cumple con lo cual la calidad del producto final se ve comprometida.

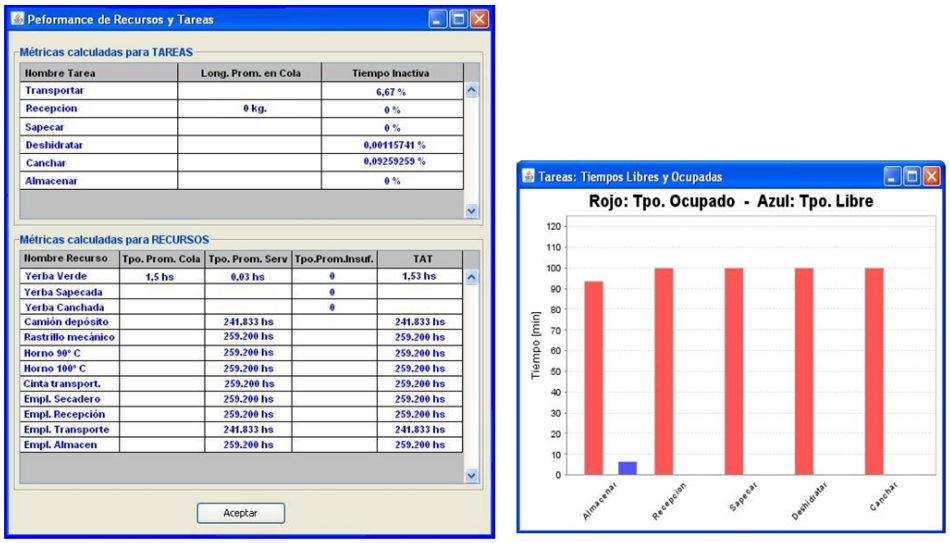
El gráfico (b) de la figura 7.10 muestra el tiempo promedio del recurso yerba verde en cola esperando ser procesado por la tarea *recepción* la cual es la primera en la versión de tarea de *secar*. Como se observa, el tiempo en cola se va incrementando dado que la tarea secar no llega a procesar todos los recursos que le llegan por día, lo cual hace que se vaya acumulando día a día.

El gráfico (c) de la figura 7.10 muestra un diagrama en barras que representa los tiempos en que las distintas tareas estuvieron ocupadas y libres. Cada tarea tiene asociada dos barras: la primera, en color rojo, indica el tiempo en que la tarea estuvo ocupada, la segunda barra en color azul, representa el tiempo ocioso de la tarea. Del análisis de este gráfico se concluye que todas las tareas estuvieron ejecutando a su máxima capacidad (no hay barras azules) excepto la tarea *transportar* que tiene un período de inactividad elevado. Esta tarea, se encarga de guardar en depósito las bolsas de 50kg

de yerba canchada para que las mismas se estacionen por un tiempo igual a dos años. El problema que se presenta en este escenario es que se requiere esperar mucho tiempo hasta obtener la cantidad necesaria de bolsas para llevarlas a depósito.

Para solucionar los problemas detectados, se elaboran propuestas que permitan mejorar la performance del proceso. Una de estas propuesta es cambiar los dos hornos que son utilizados por la tarea *sapecar* controlados manualmente, por un horno que tenga una alimentación a gas y pueda ser regulado automáticamente a través de sensores que mantenga la temperatura del horno constante, esto evitaría pérdidas en el sapecado de la hoja. La hoja verde al ser sapecada pierde el 20 % de su peso, más un 15 % que se pierde debido a los desperfecto en mantener el horno a temperatura constante. Este 15 % es recuperado usando los hornos controlados automáticamente. Otra alternativa es agregar un nuevo horno además del existente de manera de duplicar la producción de este producto intermedio. Al agregarse un nuevo horno, es necesario también incorporar otro rastrillo mecánico que alimente este horno, de esta manera, se pasa de procesar 8.000kg. de yerba verde por hora a 16.000kg por hora. Esta opción es la más conveniente y es la alternativa que se utilizó para generar un nuevo escenario y analizarlo.

El proceso se modificó en el modelo conceptual y luego se ejecuta la simulación, obteniendo nuevos resultados. La figura 7.11 muestra los resultados de ejecutar la simulación en el nuevo escenario planteado donde se utilizan dos hornos utilizados en la tarea *secar*. Se observa en la tabla de la figura (a), que la tarea *transportar* ahora tiene solo un 6,67 % de inactividad. Y el tiempo en cola del recurso yerba verde ahora es de 1,5hs. promedio por lo tanto, se mejora la calidad del producto que se obtiene cumpliendo con la restricción que se impuso en la definición del proceso.



(a) (b)

Figura 7.11: Performance del proceso modificado

7.3. Simulación Distribuida para el EM de la Cadena de Suministro

Una vez analizados los procesos de la fábrica de yerba mate, es posible incorporar este modelo en una cadena de suministro para poder interpretar las relaciones colaborativas que existen entre sus miembros. Se tomó como ejemplo la cadena de suministro mostrada en la figura 7.12 en donde la fábrica de yerba mate es uno de sus integrantes. Para este ejemplo en particular se definió el FOM (Federation Object Model) presentado en el Anexo B. Este archivo XML es un elemento clave en HLA, dado que el mismo representa los datos e interacciones que serán intercambiados por los miembros de una federación, por ejemplo el objeto *StateOrderSource* se utiza para llevar cuentas de la cantidad de órdenes que fueron procesadas en el federado que tenga asociado el rol *source*; los distintos atributos de este objeto representa los posibles estados en que se encuentra la orden que es procesada. Así por ejemplo, el atributo *delayed* registra

la cantidad de órdenes demoradas; el atributo *filled* representa la cantidad de órdenes que fueron satisfechas en tiempo y forma; y el atributo *unfilled* representa las órdenes que fueron satisfechas parcialmente. Otros objetos similares fueron definidos para los diferentes roles. Las interacciones que se definen en este archivo, están asociadas a los posibles eventos que puede generar un federado. Se representaron interacciones como: *Ask, Complaint, deliver, return*, las cuales representan las clases de interacciones, luego se definieron subclases de las mismas con sus respectivos parámetros.

Para que una federación tenga éxito, los federados deben acordar sobre el conjunto de datos a ser compartidos.

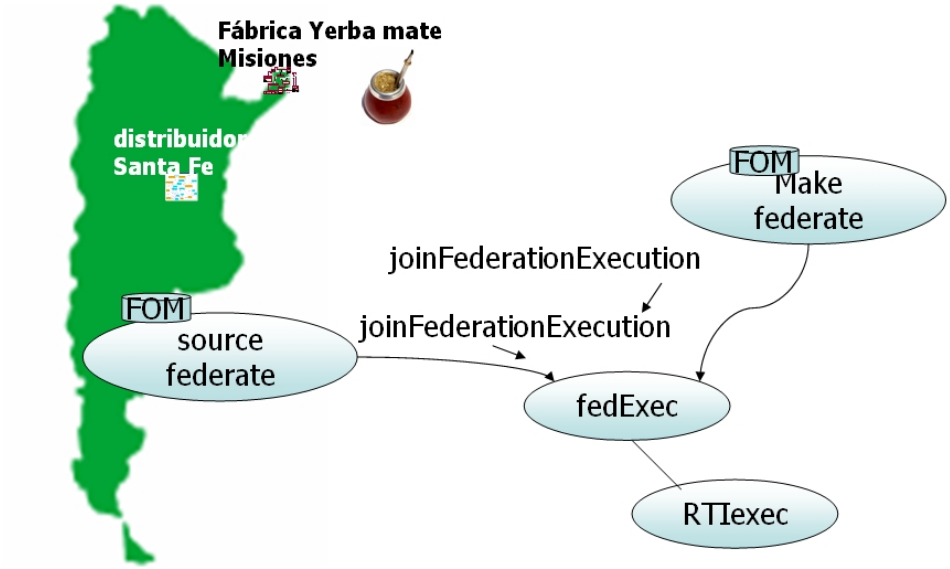


Figura 7.12: Cadena de suministro

La arquitectura DE^{2}M presentada, permite reutilizar el modelo generado para simulación local en un entorno de simulación distribuida, transformando este modelo en un federado que es capaz de sumarse a una federación, pero no tiene como función la generación de la federación.

Por esta razón, para poder armar el ejemplo y analizar cómo el federado generado con la arquitectura se comporta en la federación, se desarrolla la misma siguiendo los pasos propuestos en FEDEP (Federation development and execution process) (IEEE, 2000a) los cuales son:

A) Definir objetivos de la federación

B) Realizar el análisis conceptual

C) Diseñar la Federación

D) Desarrollar la federación

E) Planificar, integrar y testear la federación

F) Ejecutar la federación y preparar las salidas

G) Analizar los datos y evaluar los resultados

Cada uno de estos pasos será descriptos a continuación indicando los detalles de cada uno.

A) Objetivos de la federación. El **objetivo** de la federación de cadena de suministro que se crea, es el de *analizar el comportamiento de la cadenas de suministro formada por la fábrica de yerba mate y su distribuidor en Santa Fe que cooperan y colaboran en pos de alcanzar la máxima satisfacción del cliente, fijándose como deseable cumplir con las órdenes del cliente en menos de 7 días desde que la misma es recibida.*

Para ello la federación debe:

- estar formada por federados que representan los miembros de la cadena de suministro

- poder analizar la manera en que las órdenes de los clientes fluyen en esta cadena.
- poder obtener como resultado la cantidad de órdenes que sufrieron una demora mayor a 7 días en su tratamiento dentro de la cadena.

B) Análisis conceptual. El propósito del análisis conceptual es obtener un modelo del mundo real que forme parte del espacio del problema de la federación. El escenario que se plantea está formado por dos nodos correspondientes a la fábrica de yerba mate y al distribuidor. A los nodos se les asignaron los roles *make* y *source* respectivamente, de acuerdo a la descripción que se hizo en el punto 6.2.1. Estos nodos son las entidades conceptuales que forman la federación cuyos nombres son tomados de los roles que cada federado juega. En este contexto, se definen las relaciones entre estas entidades, representando las mismas los eventos que serán intercambiados cuando la federación se ejecute. Estas relaciones se muestran en al tabla 7.1. La primera columna de la tabla, representa la interacción que se genera. La segunda y tercera columna especifican los roles del origen y del destino de la misma, y la cuarta columna establece el evento que dicha interacción genera. Este evento aparecerá como una interacción en la definición del FOM. Un escenario posible se presenta en la figura 7.13 donde se observa una secuencia

Interacción	Origen	Destino	evento
Solicitar reposición mercadería	source	make	solicitarMercaderia
enviar mercadería a depósito	make	source	entregaMercadería
pagos	source	make	pagosClientes
reclamar pagos	make	source	reclamoPago
reclamar reposición	source	make	reclamoReposicion
retornar productos defectuosos	source	make	retornarDefectuoso
informar el estado del stock	source	make	estadoStock

Tabla 7.1: Interacciones entre los federados

posible de interacciones entre los nodos *make* y *source*. En principio los nodos se unen

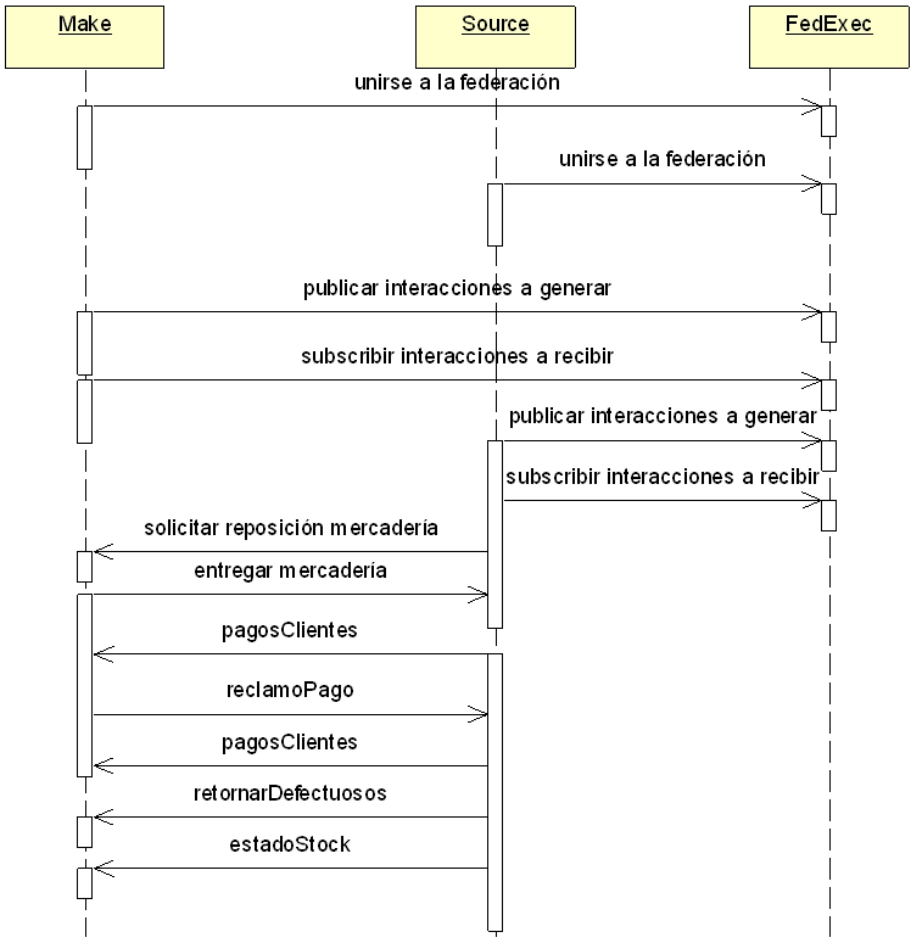


Figura 7.13: Escenario de la federación SCFederation

a la federación, luego se suscriben y publican las interacciones de interés. Una vez completado este proceso, los nodos están en condiciones de enviar y recibir interacciones, como por ejemplo, solicitar mercadería, realizar pagos, retornar mercadería defectuosa entre otras.

C) Diseño de la federación. La federación que se modela consiste en dos federados representando una fábrica de yerba mate y su distribuidor. Dado que el federado correspondiente a la fábrica de yerba mate existe como resultado del proceso de modelado explicado en los puntos anteriores y como consecuencia del uso de la arquitectura

presentada en esta tesis, falta definir el federado que corresponde al distribuidor. Este federado se diseña de manera que pueda incorporarse a la federación. Para el caso específico de este ejemplo se utiliza DE^{2}M para generar dicho federado.

D) Desarrollar la federación. Cuando una federación se crea, la misma se identifica con un nombre, el cual debe ser conocido por los federados que quieran unirse a ella. Para este ejemplo se denominó a la federación *FederacionSC* la cual está integrada por dos federados, los cuales deben tener un nombre que los identifique. El módulo federado que identifica a la fábrica es identificado con el nombre *federadoMake* y al módulo que representa al distribuidor se le asignó el nombre *federadoSource*. También es necesario definir el FOM, es decir el conjunto de objetos e interacciones que serán compartidos por los federados durante la ejecución de la federación. Este archivo XML se define de acuerdo a las entidades conceptuales definidas anteriormente. Así para cada uno de los eventos definidos en la tabla 7.1 se crea una interacción en el FOM.

Por ejemplo, a continuación se presenta una parte del FOM (presentado en el Anexo B) que define la interacción *entregarMercadería* (DeliverGoods), con la cual se especifica el envío de mercadería de un federado, que publica la interacción, a otro, que se subscribe a la misma:

```
<interactionClass name="DeliverGoods"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="SubClass of Deliver" />
   <parameter name="itemID"
      dataType="HLAreliable"
      semantics="identificador del item" />
   <parameter name="date"
      dataType="HLAreliable"
      semantics="fecha de la entrega" />
   <parameter name="quantity"
      dataType="HLAreliable"
```

```
        semantics="cantidad entregada del item" >
</interactionClass >
```

Se puede observar que la interacción tiene el nombre *DeliverGoods*, la cual será interpretada por el RTI como un mensaje *timeStamp*(el valor del atributo *order* del tag *InteractionClass* es timeStamp), es decir, cuando esta interacción sea enviada por el federado emisor, integrará la cola de mensajes de acuerdo al TSO (Time Stamp Order) en el RTI. Esta interacción se la define con tres parámetros: *itemID* representando el item o producto que se recibe, *date* identificando la fecha en que el producto es entregado y *quantity* que es la cantidad entregada del item.

Además se definen objetos que serán administrados por los federados para calcular las estadísticas. Para este ejemplo se especificaron objetos para llevar registro de las órdenes completadas en tiempo y forma, órdenes demoradas y órdenes no entregadas en cada uno de los nodos que forman la cadena. Por ejemplo, a continuación se muestra la definición del objeto *StateOrderSource* que representa la cantidad de órdenes en el nodo source (distribuidor) en sus diferentes estados, los cuales quedan representados por atributos del objeto:

```
<objectClass name="StateOrderSource"
      sharing="PublishSubscribe"
      semantics= "Clase que representa los posibles estados de las órdenes de clientes
      en el nodo source">
   <attribute name="delayed"
      dataType="integer"
      updateType="Static"
     updateCondition= "NA"
      ownership="NoTransfer"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="cantidad de órdenes demoradas en el nodo source"/>
   <attribute name="filled"
      dataType="integer"
      updateType="Static"
      updateCondition="NA"
      ownership="NoTransfer"
```

```
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="cantidad de órdenes completas en el nodo source"/>
   <attribute name="unfilled"
      dataType="integer"
      updateType="Static"
      updateCondition="NA"
      ownership="NoTransfer"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="cantidad de órdenes entregadas incompletas en el nodo source"/>
</objectClass>
```

La federación del ejemplo tiene dos federados los cuales fueron diseñados con anterioridad. A continuación se describe de que forma los federados se integran a la federación mediante la incorporación de un componente de software común. Ambos federados incluyen el método *runFederate()* (este método se encuentra definido en la clase *FederateE2M* de la arquitectura) cuyo código se presenta a continuación:

```
public synchronized void runFederate(String nfederado, String nfederacion)
throws Exception
{
    // 1.  obtener un vínculo con el RTI a través del
       RTI-Ambassador//

    miRTI = new Default1516RTIAmbassador();
    miRTI.initializeRTI(null);

    // 2. unirse a la federación //

    fedamb = new FederateAmbassadorE2M(this);
    miRTI.joinFederationExecution( nfederado, nfederacion, fedamb, null );

    // 3. TIEMPO //

    miRTI.enableTimeConstrained();
    while (!fedamb.tiempos())
```

```
            miRTI.evokeMultipleCallbacks(1.0,2.0);

        miRTI.enableAsynchronousDelivery();
        miRTI.enableTimeRegulation(new DoubleTimeInterval(10));
        fedamb.resetTiempos();
        while (!fedamb.tiempos())
            miRTI.evokeMultipleCallbacks(1.0,2.0);
        sincronizar();
    }
```

En el código se identifican tres pasos. El primer paso corresponde a obtener el vínculo con el RTI creando una instancia de la clase *Default1516RTIAmbassador* que representa el embajador RTI. Esta clase es propia del framework poRTIco. Inmediatamente después de creada la instancia, la misma se inicializa invocando al método *initializeRTI(null)*. En el segundo paso, el federado se une a la federación. Para ello se crea una instancia de la clase *FederateAmbassadorE2M(this)* la cual es parte de la arquitectura y corresponde al embajador federado. Esta clase implementa los métodos correspondientes a la funciones call back y es el nexo entre el RTI y el federado que se está generando. Luego de crearse esta instancia, el federado está listo para unirse a la federación para lo cual envía al RTI el mensaje *joinFederationExecution*. En el tercer paso, el federado debe establecer de que manera va a recibir y enviar sus interacciones y valores de atributos. En el código se observa que se envía el mensaje *enableTimeConstrined* al RTI para indicar que el federado envía interacciones en tiempo TSO. El ciclo que sigue corresponde a esperar que el RTI informe sobre la aceptación de esta solicitud por parte del federado. Luego se procede de igual manera para indicar que este federado recibe las interacciones en orden de TSO. Para ello, se invoca al método *timeRegulated* al RTI y se crea el mismo ciclo para esperar que el RTI conteste. Finalmente se invoca a un proceso de sincronización. Una vez que este punto de sincronización es alcanzado indica que los federados están listos y la federación puede comenzar a ejecutar.

E) Planificar, integrar y testear la federación. Una vez creado el FOM, se construye un nodo especializado en ejecutar la federación y calcular las métricas. Este nodo tiene como finalidad ejecutar el ejemplo que se muestra. El código necesario para crear una federación se presenta a continuación:

```
File fom = new File( "supplyChainFOM.xml" );
miRTI.createFederationExecution( "federacionSC", fom.toURL() );
nodoSource.runFederate("federacionSC", "nodoSource");
nodoMake.runFederate("federacionSC", "nodoMake");
```

Aquí se crea la federación con el nombre *federacionSC* la cual reconoce como FOM el archivo *supplyChainFOM.xml* y la integración a la federación de los federados *nodoSource* y *nodoMake*. Al ejecutar este código, los federados correspondientes a la fábrica y al distribuidor reciben el mensaje *runFederate* (el cual invocará el método presentado) con lo cual los federados se unen a la federación y se sincronizan para poder comenzar la simulación. Se testea que la federación pueda ser creada una vez que el RTI esté ejecutando en una máquina. Se testea este código por problemas que pueden existir en la interpretación del FOM, en la ejecución de la federación, la incorporación de los federados y los puntos de sincronización definidos. En este paso es importante determinar el orden en que los federados se unen a la federación y analizar la integración entre los mismos.

F) Ejecutar la federación y preparar las salidas. La federación es ejecutada, los federados se unen a ella, la simulación comienza y las salidas son obtenidas. Estas salidas corresponden a valores de ciertos objetos que han sido definidos en el FOM para mantener valores estadísticos. La ejecución de la federación se lleva a cabo como se indica en la figura 7.14 donde un usuario ejecuta el RTI y crea la federación, luego los otros federados pueden unirse a la federación para comenzar la ejecución. Se definió que

el federado correspondiente a la fábrica sea el encargado de obtener las métricas correspondientes a la evaluación de la simulación de la federación. Los datos de prueba son generados por el *ExperimentalFrame* del nodo *Source*, donde se especifica que el recurso *ordenCliente* es el disparador del proceso de simulación. El componente *Generator* de *ExperimentalFrame* genera las órdenes de clientes, luego los procesos internos de este nodo provocan la generación de interacciones *AskSupply* que serán recibidas por el federado *Make* y provocarán que el mismo active sus procesos internos. Se generaron 80.000 órdenes de cliente correspondientes a las órdenes que en promedio suceden en el transcurso de 8 meses.

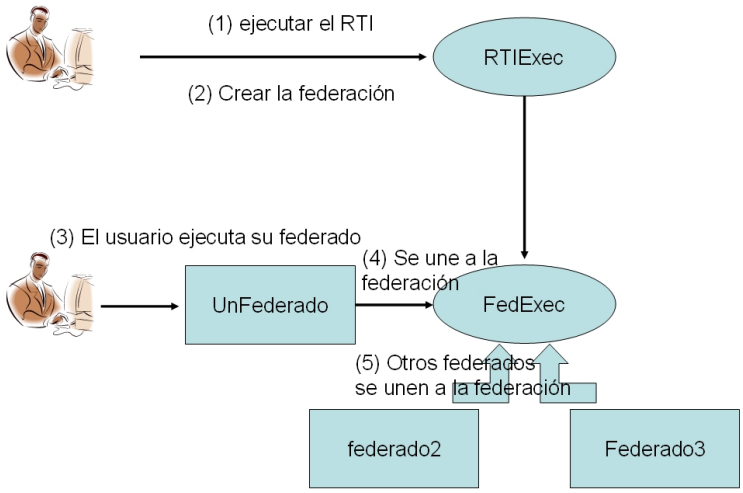


Figura 7.14: Ejecución de la federación de cadena de suministro

G) Analizar los datos y evaluar los resultados. Un ejemplo de los datos que se pueden obtener de la simulación se muestran en la figura 7.15. A partir del análisis surgen interpretaciones del proceso que se ejecutó, por ejemplo, se observa que el 73 % de las órdenes del cliente fueron satisfechas en tiempo y forma, mientras que un 18 % sufrieron demoras en la entrega, un 9 % no pudieron ser cumplidas y un 27 % de las órdenes tuvieron algún inconveniente para ser procesadas. Esto habla de problemas

en la distribución y la entrega de los productos. Dado que se trata de un producto sustituible, el porcentaje de órdenes con problemas es muy elevado y una mejora debe ser planteada en el proceso.

A partir de esta evaluación, es posible introducir mejoras o cambiar políticas. Nuevos escenarios pueden plantearse y nuevas simulaciones ejecutarse. Por ejemplo podría proponerse un cambio en los volúmenes de stock mantenidos en el distribuidor de manera de acelerar el proceso de entrega y evitar órdenes incumplidas por falta de mercadería.

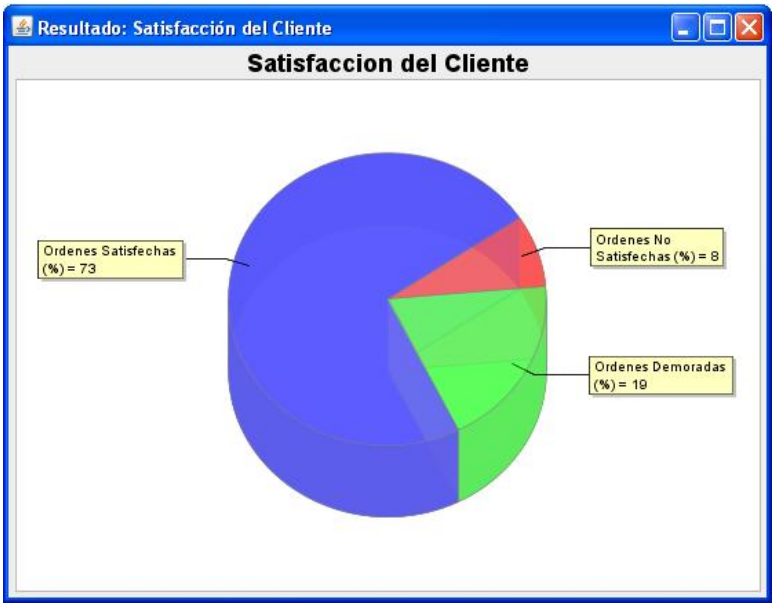


Figura 7.15: Resultados de la simulación de la cadena de suministro

7.4. Conclusiones

En este capítulo se desarrolla un ejemplo que muestra el uso de la arquitectura. Se focaliza en la empresa de producción de yerba mate, para la cual se muestra paso a paso la construcción del modelo de empresa conceptual. A partir de las diferentes vistas propuestas, se presentan los diagramas y se ejecuta la simulación local mostrando los

resultados los cuales se utilizaron para introducir mejoras en los procesos internos de esta empresa. Luego, se describe una cadena de suministro donde esta empresa participa. Para el desarrollo de la federación, se siguieron los pasos propuestos en FEDEP donde se definen los objetivos, integrantes y el FOM que va a ser utilizado por los federados. Luego, la federación es creada y la simulación distribuida es ejecutada produciendo valores cuantitativos que permiten analizar la satisfacción del cliente en función de las órdenes completas en tiempo y forma por la cadena de suministro representada por la federación. A partir de estos datos es posible introducir mejoras en los procesos de los diferentes miembros para mejorar la performance de toda la cadena.

8 Conclusiones y Trabajos Futuros

8.1. Conclusiones

El concepto de modelo de empresa ejecutable y distribuido fue el punto de partida en la elaboración de la arquitectura propuesta. Inicialmente se trabajó sobre los denominados modelos de empresa, resaltando la necesidad de un lenguaje que permita plasmar el conocimiento de la misma. Así, surge la adopción del lenguaje Coordinates. Del análisis del mismo, se observa que los modelos de empresa pueden ser generados de una manera modular con un proceso interactivo e incremental. A partir de aquí se especifica y define la primera capa de la arquitectura donde se identifican los elementos necesarios para el diseño y análisis de los modelos de empresa.

Se propone para la construcción de los modelos, distintas vistas que permiten separar o visualizar en forma específica cada dimensión de análisis. Así, la vista funcional ayuda a comprender las actividades llevadas a cabo en una organización, y concentrar la atención sobre aspectos relacionados con el orden de ejecución de las tareas y la descomposición de las mismas en tareas más simples. Una implementación de esta capa se desarrolla dando origen al prototipo Coordinates Workbench. Posteriormente el mismo se extiende con la incorporación de la vista dinámica, donde se presentan los recursos

y los cambios de estados originados por la participación de estos en las tareas.

A partir de estos modelos, la empresa puede analizarse desde sus aspectos estáticos, sin embargo la necesidad de analizar la dinámica de la misma se hizo una necesidad cada vez más creciente. Así, se comienza a trabajar sobre la incorporación de la capacidad de simulación al modelo conceptual de empresa. Se analizan los diferentes enfoques disponibles de los cuales se seleccionó el modelo DEVS para la implementación de la simulación en procesos de empresa. Este paradigma se selecciona por poseer características que permiten construir modelos en forma modular y jerárquica, y disponer de una interfaz bien definida. Los primeros estudios llevaron a desarrollar modelos DEVS especiales con los cuales poder representar los distintos conceptos del modelo conceptual de empresa. Estos conceptos se presentaron en base a los bloques de construcción del modelo de simulación identificando puertos, eventos, funciones de transición de estados internas y externas, funciones de salidas y función del tiempo.

Luego, fue necesario mostrar que con estos bloques de construcción de modelos de simulación, definidos específicamente para procesos de empresa, era posible representar un modelo equivalente al expresado usando el lenguaje Coordinates. Así, se desarrolla un ejemplo, donde se puede verificar que todos los conceptos que intervienen en el modelo de empresa, expresado en el lenguaje Coordinates, tienen su equivalente en los modelos DEVS definidos en esta tesis. De esta manera se logra un ambiente integrado de diseño y análisis tanto estático como dinámico de los modelos de empresa. En este contexto el término *integrado* identifica la generación del modelo de simulación a partir del modelo conceptual de la organización. Si bien, hasta ese momento no se identifica el cómo el modelo podía ser transformado, se deja en claro que la existencia de esa equivalencia entre los conceptos que intervienen en ambos modelos, posibilita la automatización de esa transformación.

La arquitectura fue verificada con un caso de estudio consistente en una fábrica de

producción de yerba mate, para la cual se desarrollan los modelos conceptuales y a partir de estos, se genera el modelo de simulación. Esta experiencia permitió entender cuales eran las reglas necesarias para realizar la transformación del modelo conceptual al modelo de simulación utilizando el vocabulario definido con los modelos DEVS especializados.

Seguidamente, se identifican las reglas de transformación de modelos las cuales se implementan dando lugar a la automatización requerida en la obtención del modelo de simulación a partir del modelo conceptual de empresa. La automatización lograda, tiene dos ventajas importantes: evita pérdida de información y facilita la tarea al experto en procesos de negocios ocultando los conceptos de simulación.

A continuación se identifica la necesidad de contar con la posibilidad de construir modelos para ambientes distribuidos. Este requerimiento puso en evidencia que era imprescindible disponer de una plataforma que permitiera integrar a los diferentes modelos de simulación desarrollados localmente. Para tal fin se adopta el estándar HLA 1516-2000 como base para la interoperabilidad sintáctica de los simuladores y se adapta el ciclo de simulación DEVS de los simuladores locales para que los mismos pudieran formar parte de una federación HLA. Debido a que DEVS propone separar modelos de las máquinas de simulación que interpretan esos modelos, se continúa con esa filosofía generándose una máquina de simulación que permitiera interpretar un modelo DEVS bajo un entorno distribuido basado en HLA. Por otro lado la interoperabilidad semántica fue tenida en cuenta proponiéndose para ello el uso de roles que encapsulan el tratamiento de la interpretación de las interacciones entre federados HLA.

La arquitectura propuesta que describe la integración entre modelado y simulación de modelos de empresa, da soporte no sólo a la descripción de un modelo de empresa sino también al análisis del mismo a través de simulación. Por un lado el uso de un lenguaje orientado a negocios como lo es Coordinates, y las vistas que el mismo presenta, es una

característica útil que permite describir un modelo de empresa empleando conceptos conocidos, lo cual facilita ostensiblemente la actividad. Por otro lado, la generación automática del modelo de simulación integra todas las vistas, permitiendo analizar la empresa desde el punto de vista de su comportamiento dinámico. Esta facilidad tiene consecuencias inmediatas:

- los resultados del análisis pueden servir de base para procesos de toma de decisión ya que se están evaluando todos los aspectos de la empresa,
- mantiene la consistencia entre las vistas,
- presenta una visión integrada del modelo.

El uso de la arquitectura acentúa su importancia cuando se trata de empresas distribuidas, en red o cadenas de suministro, donde la construcción modular de los modelos de empresa ejecutables representa una gran ventaja. Esta facilidad permite reducir los costos y tiempo necesarios para generación de modelos, evitando concentrar toda la información de la empresa en un solo lugar, facilitando la depuración, corrección y modificación de los módulos que conforman el modelo. Además no es necesario exhibir información confidencial ya que cada módulo es visto como una caja negra con una interfaz bien definida y acordada entre los integrantes.

Por lo tanto puede resumirse como principal contribución de esta tesis la propuesta de una arquitectura que permite integrar modelos de empresas, definidos en un lenguaje conceptual, con sus correspondientes modelos de simulación, los cuales son generados a partir de los primeros en forma automática. A esta estructura de modelos se le adiciona la posibilidad de poder asociarse a una federación distribuida de modelos de simulación, lo cual facilita la integración de modelos complejos.

8.2. Trabajos Futuros

8.2.1. Utilización de Estructuras Variables en Simulación

El progreso incremental de la tecnología basada en componentes en la ingeniería del software, ha permitido desarrollar entornos flexibles donde sus mayores ventajas son el reuso y la construcción de modelos al estilo de un rompecabezas. Este paradigmas fue ampliamente usado en la arquitectura presentada en esta tesis, lo que origina la posibilidad de crear estructuras variables de los modelos de simulación en tiempo de ejecución.

La incorporación del manejo de estructuras variables de modelos en la arquitectura fue evaluada durante su desarrollo, sin embargo no fue definida y se presenta como un trabajo futuro.

Si se analiza la estructura que presenta un modelo de empresa generado con Coordinates, se observa que una tarea compuesta puede tener asociada múltiples descomposiciones o versiones de tareas. Estas descomposiciones representan diferentes formas de llevar a cabo una tarea, y dependiendo de ciertas condiciones (estado de recursos, ejecución de otras tareas, etc) se puede adoptar una u otra forma. Ahora bien, durante una simulación las condiciones pueden ir cambiando y con esto, las versiones con que una tarea puede ejecutarse.

En la arquitectura propuesta se identifica una tarea asociada a una única versión, a la que se identificó como *default* y esa versión es la que se utiliza durante todo el tiempo de simulación. Sin embargo se podría pensar que durante la ejecución esa versión asociada pueda ir cambiando de acuerdo a condiciones evaluadas.

El *System Entity Structure* (SES) (Hu y otros, 2005) provee una manera de especificar composición con información sobre descomposición, acoplamientos y taxonomías,

representa un marco formal para describir una familia de estructuras.

Desde el punto de vista de los DE^{2}M, SES representa el espacio de problemas con las diferentes configuraciones. A partir de este espacio de problemas, es posible identificar una estructura particular a través del proceso de *poda*. Así se deja abierta la posibilidad de incorporar un componente a la arquitectura llamado *Puzzle* encargado de armar la familia de modelos utilizando SES/MB donde en la base de modelo (MB) se manipulen las instancias de los componentes que intervienen en la estructura. Ese nuevo componente, es el encargado de obtener el modelo sintetizado que debe ser ejecutado.

Se puede notar que a medida que las condiciones cambien, un nuevo modelo debe ser obtenido de acuerdo a las restricciones que se establecieron en la familia de modelos. Para ello el componente *Puzzle* debe contar con un agente inteligente que perciba el modelo actual y sus cambios y proponga nuevos cambios a la estructura que está actualmente ejecutando. La figura 8.1 representa la meta estructura de un modelo de simulación generado con esta arquitectura.

Así, en tiempo de ejecución, los recursos modifican sus estados y las condiciones con relación a la versión de tarea que está actualmente en ejecución, cambiando la estructura del modelo de simulación. Entonces, un agente monitorea la situación. Toma como percepción los estados de los recursos, evalúa las restricciones y determina los cambios que son necesario en la estructura del modelo de simulación. En este contexto los posibles cambios son: (i) agregar un modelo, (ii) eliminar un modelo. Debido a que la interfaz de la tarea no cambia no es necesario tener en cuenta cambios como agregar o eliminar puertos. La figura 8.2 muestra la estructura definida para el componente *Puzzle*.

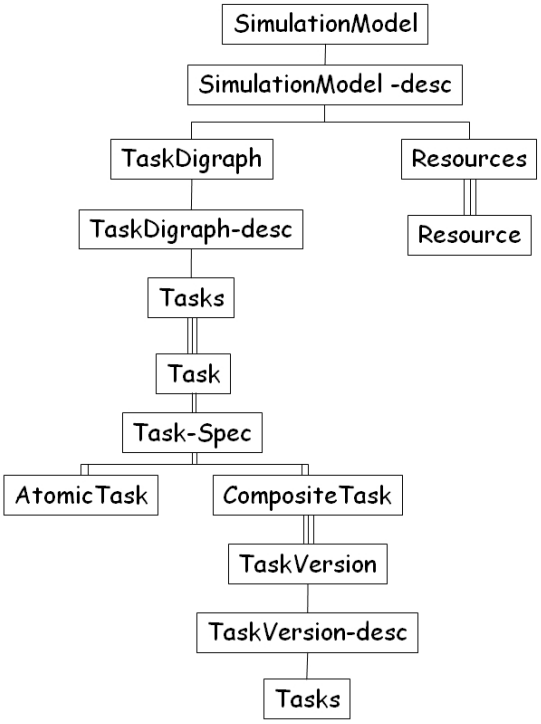


Figura 8.1: Meta estructura de un modelo de simulación

8.2.2. Desarrollo de la Federación y Contribuciones en la Composición de Federados

Uno de los grandes problemas que afecta considerablemente el uso de simulación distribuida es sin duda la composición de los diferentes simuladores. Esta composición se refiere al intercambio de datos y control entre simuladores, la causalidad entre las interacciones de entrada y salida, y la sincronización de la ejecución con respecto al tiempo. HLA provee de interoperabilidad entre simuladores, sin embargo la composición es altamente dependiente del modelo de datos o FOM (Federation Object Model) utilizado en la federación. En otras palabras, HLA provee una clara definición de la interfaz necesaria para que los federados interactúen en una federación, pero el entendimiento a nivel semántico de los datos intercambiados no está resuelto por este estándar y corresponde al problema de la composición de los simuladores.

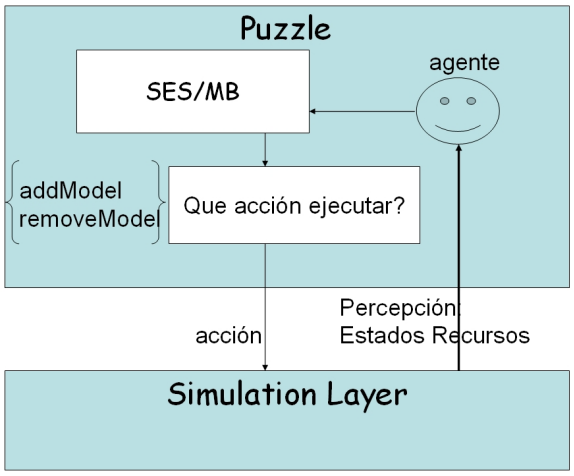


Figura 8.2: Estructura del componente Puzzle y su interacción con el coordinador

Como trabajo futuro, se prevé el desarrollo de un *servidor de federaciones*, herramienta que permita diseñar federaciones siguiendo los pasos propuestos en FEDEP y que contribuya a lograr la composición de los diferentes simuladores dentro del contexto de modelos de empresa.

El diagrama de la figura 8.3 muestra las funcionalidades que esta herramienta debe proporcionar.

Los casos de uso 1. y 2. corresponden a la creación de una federación, esto involucra definir su nombre y los principales objetivos. Los distintos federados que pueden intervenir serán registrados indicando nombre y dirección del mismo.

El caso de uso *3. Crear modelo conceptual* es sin duda el paso fundamental para lograr la composición. El resultado que se espera lograr es la obtención del BOM (Base Object Model) (SISO-STD-003-2006, 2006) para federados y federaciones. Un BOM es un modelo de objeto conceptual que puede tener diferentes implementaciones asociadas. Una implementación de un BOM puede ser un archivo XML correspondiente a un FOM para una federación particular. Una característica que tiene este modelo de objetos es que puede ser construido en forma modular. De esta manera, los diferentes BOM pueden

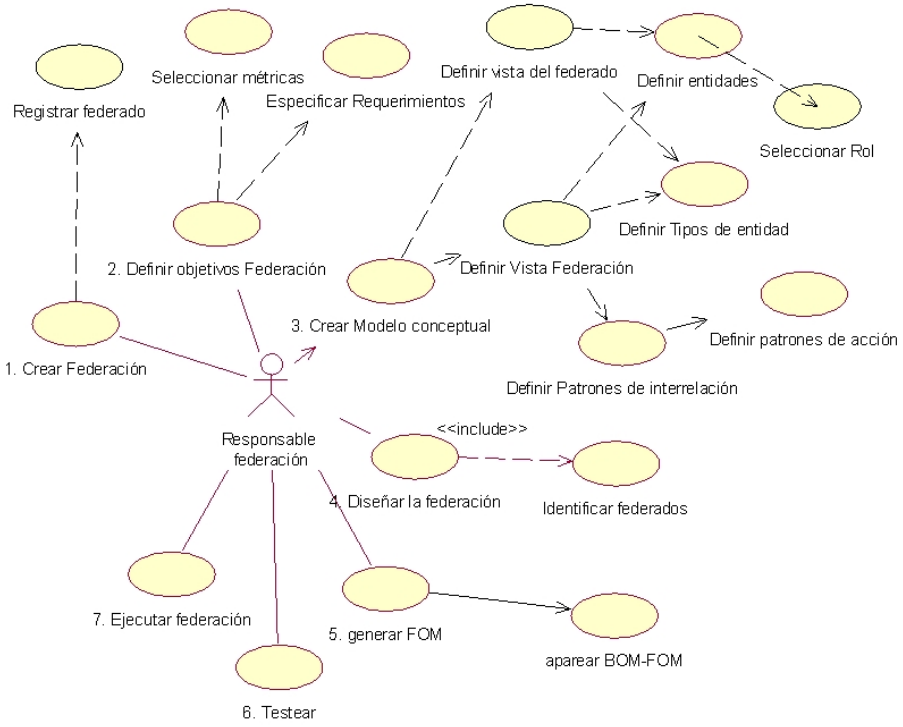


Figura 8.3: Diagrama de casos de uso de un Servidor de federaciones

ensamblarse generando un nuevo BOM el cual es llamado Assembly BOM.

El modelo conceptual de la vista del federado, permite a un federado identificar el conjunto de entidades conceptuales, eventos, estados y patrones de interacción entre entidades que pueden ser administrados por el mismo. Esto da como resultado el BOM del federado, correspondiente al modelo conceptual de objetos del mismo. En este paso, se identifican las relaciones que existen entre los elementos que forman parte del BOM con los objetos, interacciones, parámetros y atributos HLA definidos en un FOM o SOM correspondiente al federado.

El modelo conceptual de la vista de la federación, permite a una federación obtener su modelo conceptual de objetos componiendo o ensamblando las definiciones hechas con anterioridad por los federados. De esta manera, es posible obtener un modelo de objetos aún cuando todavía no se han identificado los federados que van a participar

en un escenario dado (esto corresponde al caso de uso 4. diseñar la federación).

El caso de uso *4. diseñar la federación*, permite definir el escenario identificando el conjunto de federados que participarán.

Luego, el caso de uso *5. generar FOM*, es el responsable por obtener el FOM que será utilizado en la ejecución de la federación. La generación de este archivo se hace a partir del apareamiento entre las entidades conceptuales que fueron definidas en el BOM con las entidades que pertenecen al FOM. Por ejemplo, en este proceso se asocia una entidad conceptual de BOM con una clase de Objeto HLA perteneciente al FOM.

Los casos de uso *6. Testear* y *7. Ejecutar federación* tienen como propósito verificar que la federación puede ser ejecutada en cuanto a disponibilidad de recursos de red y de procesamiento. Finalmente para ejecutar la federación, se debe ejecutar el RTI e inicializar los federados para que puedan unirse a la federación. El nombre asignado a la federación en el paso 1, es utilizado para crear la ejecución de la federación y dado a conocer a los federados para que estos puedan unirse.

A

Coordinates Workbench

Coordinates Workbench es un prototipo desarrollado basado en la arquitectura DE^{2}M que permite el diseño y ejecución de modelos de empresa. Coordinates Workbench se implementó en Java y utiliza dos framework como bibliotecas: CoreDevs y Portico.

La ventana principal de esta aplicación se muestra en la figura A.1. Esta ventana está dividida en dos partes principales: el panel de la izquierda muestra la vista del repositorio, y el panel de la derecha es el área de diseño.

En la vista del repositorio, se muestran todas las definiciones que se realizaron como ser por ejemplo las tareas, sus versiones y los recursos en una estructura de árbol. Este panel a su vez tiene tres posibles formas de mostrar las definiciones, cada una está representada por las pestañas (o tabs) que aparecen en este panel: (i) *TareaxRecursos*, (ii) *RecursosxTareas* y (iii) *VersionesxTareas*. La primera *TareasxRecursos*, identifica para cada recurso cuales son las tareas en las cuales ese recurso está involucrado. La segunda, *RecursosxTareas*, identifica para cada tarea el conjunto de recursos que participan. Finalmente *VersionesxTareas* identifica la descomposición de tareas en subtareas a través del concepto Versión de tareas.

El área de diseño representa el especio donde se irán construyendo los distintos

modelos. Dependiendo del modelo, tendrá una barra de herramientas asociada que permite identificar los conceptos apropiados para ese modelo.

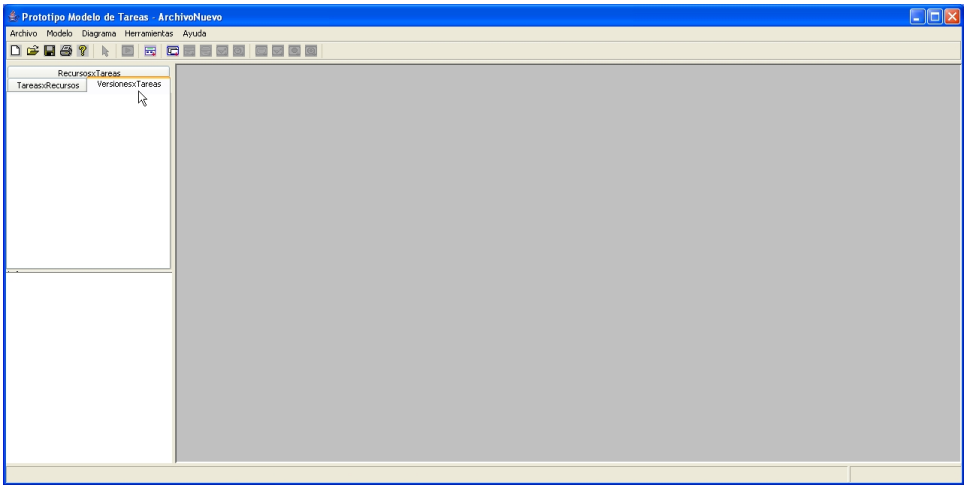


Figura A.1: Coordinates Workbench: Ventana principal

Crear un proceso

Un proceso representa la tarea de mayor nivel en la jerarquía de descomposición de tareas. En una construcción del modelo conceptual de empresa top-down se comenzará especificando las tareas procesos que se desean definir. Para crear un proceso se selecciona del menú principal la opción *Diagramas/crear/proceso*, con lo cual el proceso se crea como raíz de descomposición de las tareas. La figura A.2 muestra el cuadro de diálogo que aparece al seleccionarse la opción antes mencionada. Una vez que el proceso fue creado, se puede definir para el mismo una versión de tarea que identifique la forma en que dicho proceso es descompuesto en tareas más simples. Para ello, sobre la vista del repositorio se puede presionar el botón derecho del mouse sobre el proceso al que se le quiere agregar una versión. Un menú desplegable aparece del cual debe seleccionarse la opción *Agregar Versión* como se muestra en la figura A.3. Al agregarse una versión de tarea aparece sobre el panel que está a la derecha un área de dibujo donde es posible

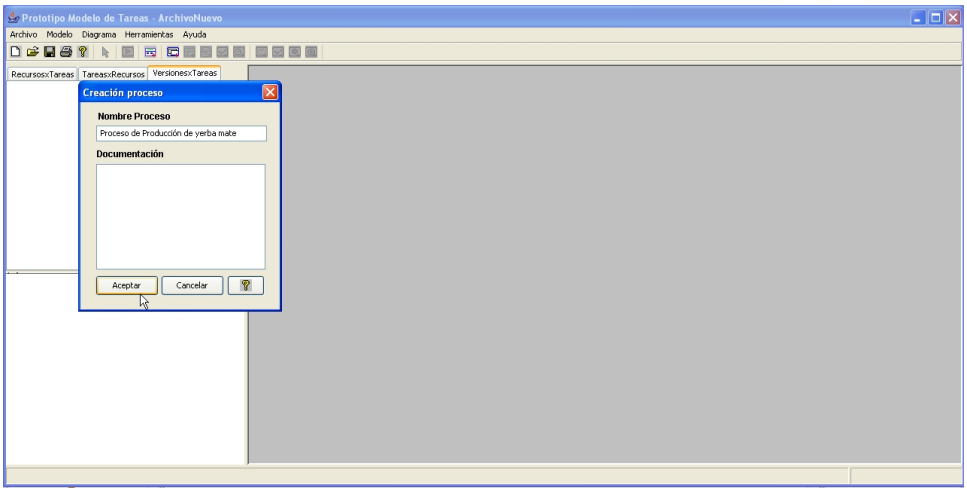


Figura A.2: Crear un nuevo proceso

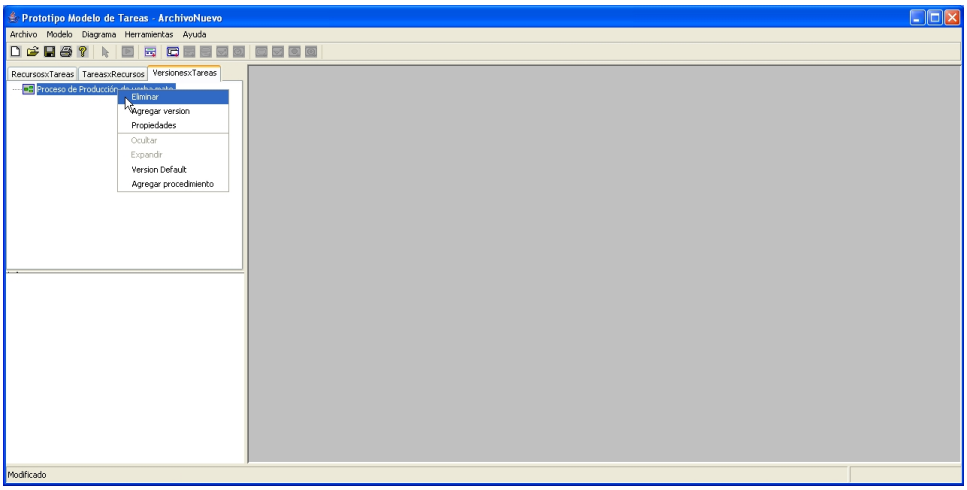


Figura A.3: Agregar una Versión de tarea a Proceso

comenzar a dibujar la versión, incorporando tareas, recursos y relaciones entre estos elementos. Sobre el área de dibujo se activa una barra de herramienta que facilita el acceso a las acciones posibles para definir la versión de tarea. La figura A.4 muestra la ventana del workbench correspondiente a la definición de una versión de tarea. En las

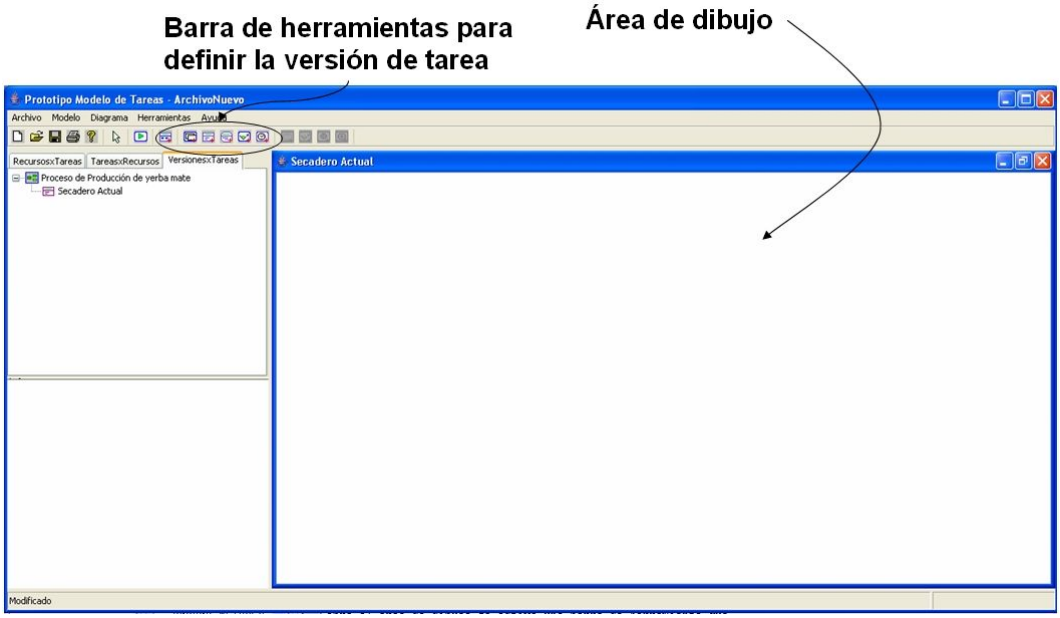


Figura A.4: Definición de la Versión de tarea Secadero Actual

próximas secciones se analiza la creación y utilización de los elementos que intervienen en los distintos diagramas.

Crear Recursos

Para crear recursos se puede utilizar la opción *Diagrama/crear/Recurso* con lo cual aparece la ventana de diálogo que se muestra en la figura A.5 en donde es posible escribir el nombre de un recurso y agregarlo a una lista de recursos creados. Estos recursos formarán parte de la vista del repositorio. Por ejemplo en la figura se observa que se han creado dos recursos: *Documento Comercial* y *Factura*. Ambos aparecen en

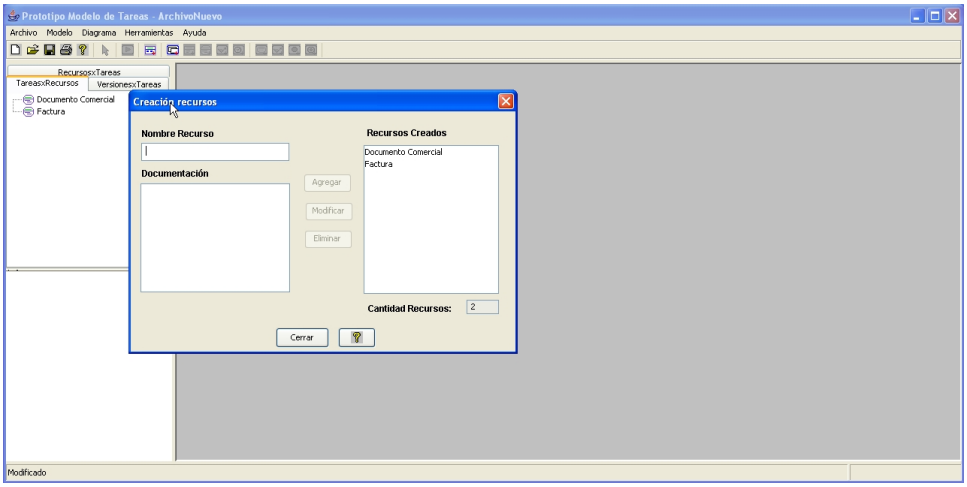


Figura A.5: Ventana de diálogo: Creación de recursos

el panel de la izquierda que representa la vista del repositorio.

Es posible también crear nuevos recursos a partir del diagrama de recursos como se muestra en la figura A.6. Para ello es necesario crear un diagrama de recursos o abrir uno existente. Luego desde la barra de herramienta se selecciona el icono correspondiente al recurso para incorporar un nuevo recurso. En la figura A.6, por ejemplo, se observan los recursos creados hasta el momento. Los recursos definidos, van a participar de la

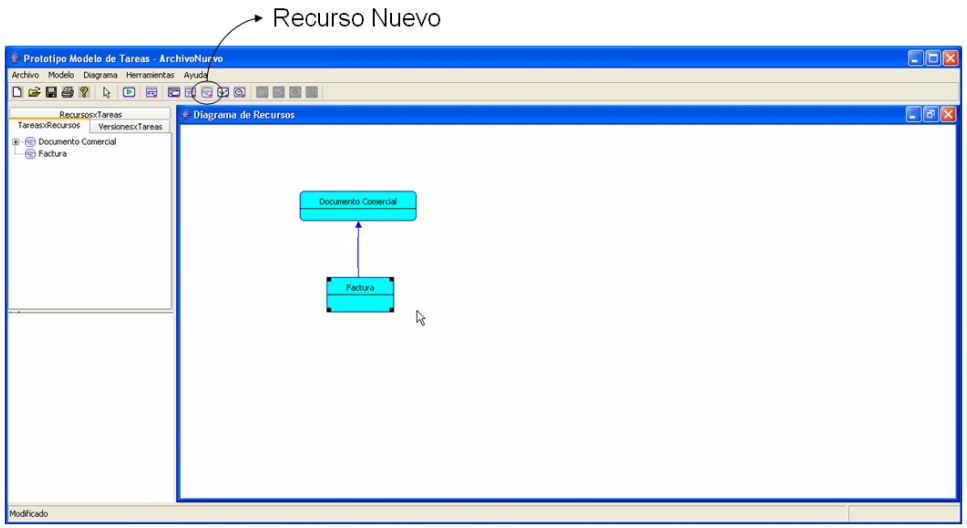


Figura A.6: Diagrama de Recursos

definición de tareas y las instancias de los recursos participarán de las versiones de tareas.

Existen diferentes formas de crear una instancia de un recurso. Cuando una tarea es seleccionada de la vista del repositorio, y arrastrada al área de dibujo correspondiente a una versión, se crean instancias de los recursos que participan de esa tarea. Otra posible forma es seleccionar el icono correspondiente a un nuevo recurso de la barra de herramientas, asociada a la definición de la versión de tarea con lo cual aparece un cuadro de diálogo como el que muestra la figura A.7 donde es posible seleccionar un recurso de los definidos hasta el momento e indicar el nombre de la instancia o la palabra *indefinido* para indicar que podría ser cualquier instancia del recurso. En este último caso en el diagrama de tareas, la instancia del recurso aparece con el mismo nombre que su clase. Por ejemplo, en la figura A.7 se creó una instancia del recurso

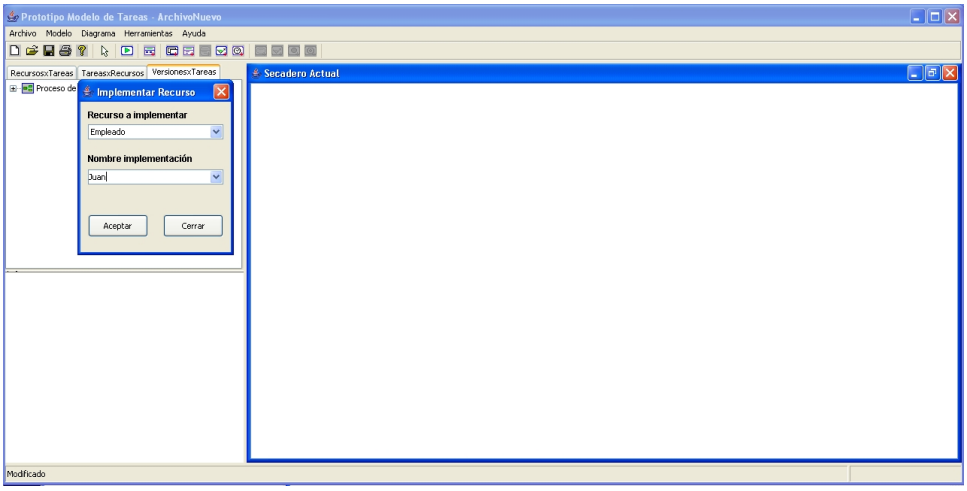


Figura A.7: Instancia de un Recurso

Empleado asignándole el nombre de instancia *Juan*.

Crear Tareas

Para crear una tarea existen dos formas posibles: (i) crear la tarea desde la vista del dominio o (ii) crear la tarea desde la versión de tareas. En el primer caso la tarea se define conceptualmente indicando los recursos que participan en la misma como se muestra en la figura A.8.

La definición de la tarea involucra definir la misma indicando nombre, duración y descripción como se muestran en la figura A.8 (a). Seguidamente, se debe identificar el o los recursos que participan en esta tarea y los tipos de relaciones entre ambos. La figura A.8 (b) muestra la ventana donde se seleccionan los recursos y las relaciones. Por ejemplo, en la figura A.8 (b) la tarea creada tiene una relación *usa* con el recurso *Empleado*. Existen dos listas desplegables, una para seleccionar el tipo de relación y la otra para seleccionar los recursos definidos hasta el momento. En caso que el recurso no haya sido definido aún existe la opción de generarlo seleccionando el botón *Nuevo Recurso*, que se encuentra en la parte inferior izquierda de esta ventana con lo cual se accede a la ventana de creación de recurso mostrada en la sección anterior. La otra forma

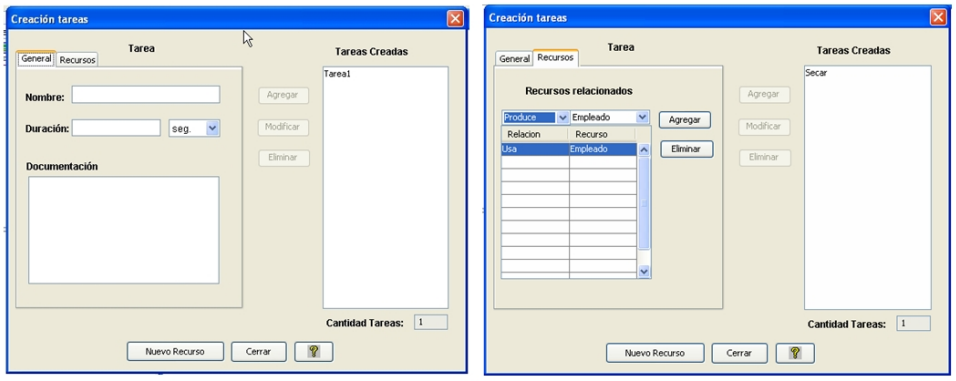


Figura A.8: Crear una tarea en la vista del Dominio

de crear una tarea es directamente en la versión de tareas. Cuando se está describiendo

una versión es posible arrastrar las tareas que aparecen en la vista del repositorio o bien seleccionar el icono de *Tarea nueva* de la barra de herramientas. La figura A.9 muestra la creación de dos tareas, *recepción* y *sapecado*, formando parte de la versión de tarea *Producción de yerba mate* A medida que las tareas se crean, aparece su vista en el panel

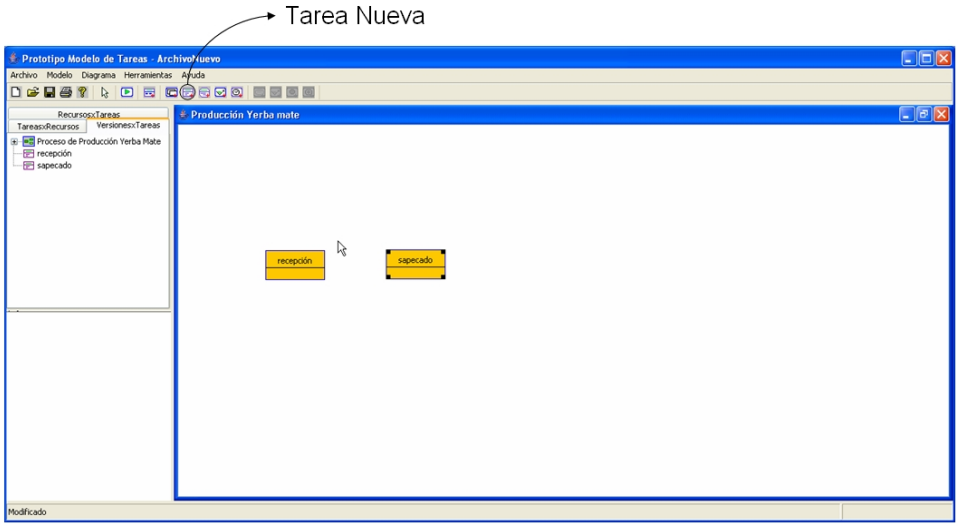


Figura A.9: Crear una tarea desde la ventana de Versión de Tarea

de la izquierda que muestra la jerarquía de las mismas. Así, las tareas que participan de una versión son representadas como sub-tareas de la tarea de mayor nivel para la cual la versión se define.

Una vez que las tareas se crean, y participan de versiones, es posible cambiar las propiedades de las mismas. Para ello presionando el botón derecho del mouse sobre la vista de la tarea, se accede a una ventana de diálogo como la que se muestra en la figura A.10, donde se puede asignar la duración de la misma. Es posible también especificar las propiedades de las relaciones establecidas entre tareas y recursos. Para ello es necesario acceder al menú desplegable que aparece al presionar el botón derecho del mouse sobre la relación. De esta forma aparece la ventana de diálogo mostrada en la figura A.11. En este caso corresponde a la relación *produce* entre la tarea *recepción* y el recurso *yerba*

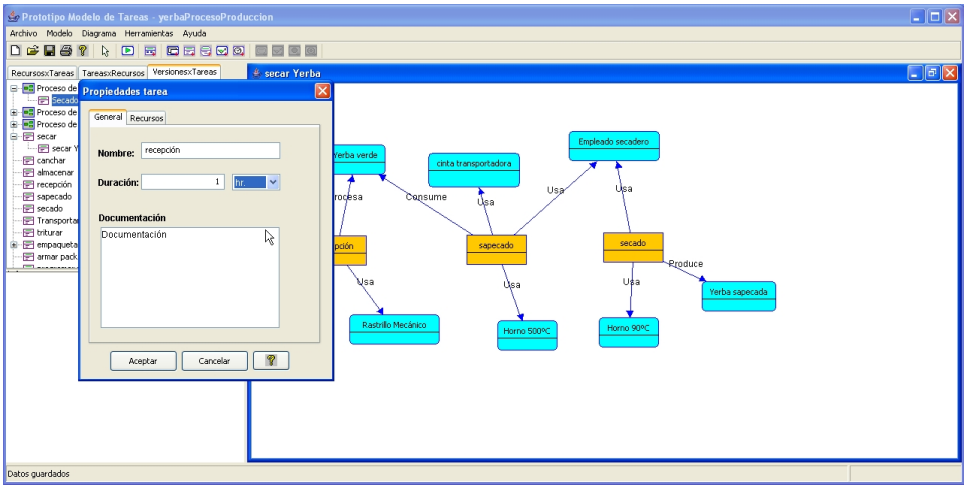


Figura A.10: Propiedades de la tarea

verde, donde es posible identificar la cantidad de ese recurso que es producido por la tarea.

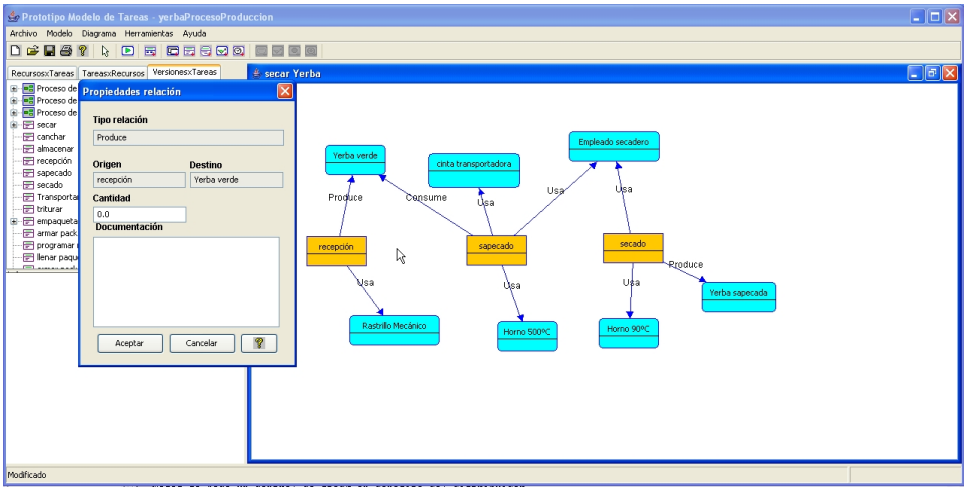


Figura A.11: Propiedades de la relación Tarea - Recurso

Crear ciclos de vida de recursos

Los recursos tienen asociado un ciclo de vida que corresponde a la vista dinámica. Para generar el ciclo de vida asociado a un recurso es necesario seleccionar el recurso,

ya sea desde la vista del repositorio o de algunas de sus vistas que aparecen en versiones de tareas. Al presionar el botón derecho del mouse, se despliega un menú del cual se debe seleccionar la opción *crear ciclo de vida*. Un área de dibujo vacío aparece. La barra de herramientas que se activa corresponde a los elementos necesarios para el ciclo de vida: nuevo estado inicial, nuevo estado final, nuevo estado, transiciones. Los estados compuestos se crean al incorporar un estado dentro del área de dibujo del otro. Para crear un estado ortogonal, el estado debe ser seleccionado y elegirse la opción asociar ciclo de vida del menú desplegable que aparece al presionar el botón derecho del mouse sobre el estado.

La figura A.12 muestra el ciclo de vida para el recurso *Empleado*. En este caso se definieron dos estados: libre y ocupado. Para identificar las tareas que generan la transición, se debe presionar el botón derecho sobre la transición con lo cual aparece la ventana de diálogo que se muestra en la figura A.13 donde es posible seleccionar una tarea de la lista de tareas. Esta lista se genera a partir de todas las relaciones tareas-recursos asociadas al recurso en cuestión. Una vez que la tarea se selecciona se debe indicar si el evento correspondiente a la transición que se está definiendo corresponde al inicio de la tarea o a la finalización, para ello se selecciona una de las tres opciones que aparecen: *inicio de la tarea*, *fin de la tarea*, *sin estado intermedio*. Esta última opción corresponde seleccionarla cuando ninguna tarea es asociada a la transición.

Ejecutar el modelo

Una vez que el modelo conceptual se concluye es posible ejecutar el modelo localmente o crear el federado para que se una a una federación. En la barra de herramientas aparece un botón con una flecha color verde que representa la acción ejecutar. Una vez que un proceso se selecciona, es posible presionar sobre el botón ejecutar para poder

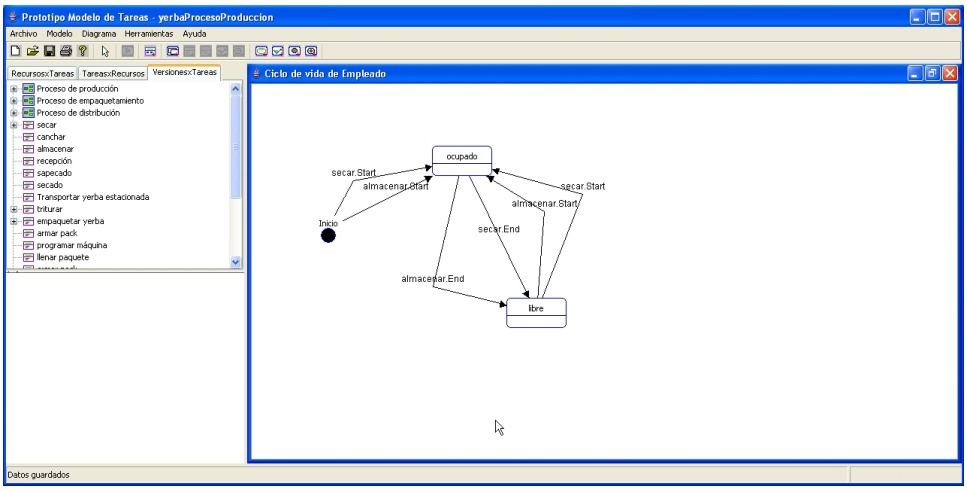


Figura A.12: Ciclo de vida del recurso Empleado

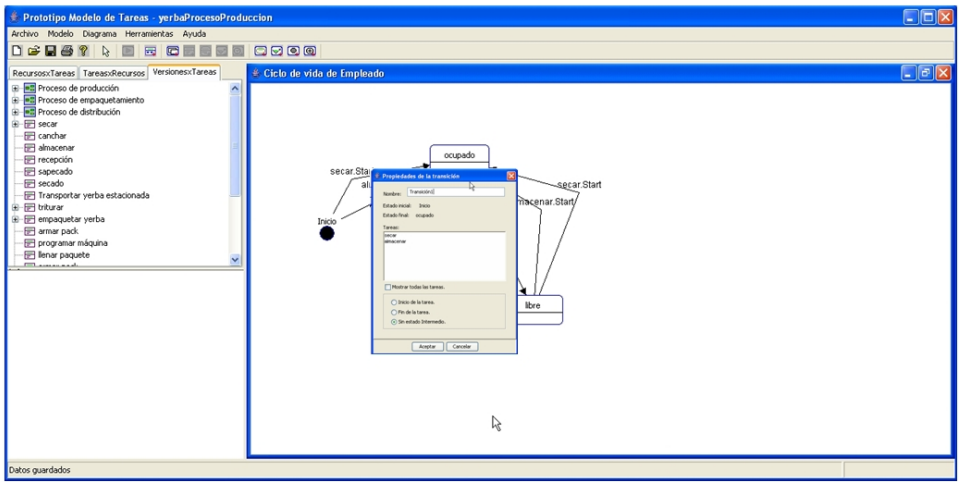


Figura A.13: Propiedades transición de estado

acceder a la ventana de diálogo de simulación. Para el primer caso, es necesario identificar las condiciones de finalización de la simulación como por ejemplo el tiempo de simulación como se muestra en la figura A.14. Para el caso de la simulación distribuida,

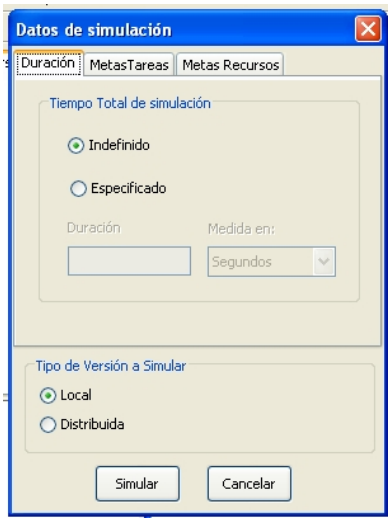


Figura A.14: Datos para la simulación local

se espera que el RTI esté en ejecución en alguna de las máquinas conectadas a la red y el FOM correspondiente a los roles esté disponible localmente. Luego al crearse el federado, el mismo ejecuta un código para unirse a la federación.

B

FOM

El principal propósito de un HLA FOM (Federation Object Model) es proveer una especificación para el intercambio de datos entre federados en un formato común. Los datos incluidos en este archivo, corresponden a una enumeración de todas las clases de objetos e interacciones pertinentes a la federación, y una especificación de atributos y parámetros que caracterizan estas clases. Es importante destacar que este archivo es considerado un contrato del modelo de información que es necesario, pero no suficiente para alcanzar interoperabilidad en la federación. El archivo se presenta con el formato de un archivo XML con elementos para definir: (i) la colección de clases de objetos (tag <*objects*>); (ii) la colección de interacciones (tag <*interactions*>); (iii) las posibles dimenciones (tag <*dimensions*>); (iv) las posibles formas de transporte (tag <*transportations*>) y (v) los tipos de datos válidos en el FOM (tag <*dataTypes*>).

Una clase de objeto HLA es una colección de objetos que tienen características o atributos comunes. La estructura de una clase de objeto HLA puede ser definida jerárquicamente. Las subclases se consideran especializaciones de las superclases. Las subclases siempre heredan los atributos de su superclase y solo se soporta herencia simple. Una clase de objeto se define a través del tag <*objectClass*>seguida por el nombre de la clase (atributo *name* del elemento *objectClass*) y la información de las capacida-

des de publicar o suscribir dicha clase (atributo *sharing* del elemento *objectClass* en el archivo XML), los posibles valores para este último son: *Publish, Subscribe, Neither,* y *PublishSubscribe*. La clase *HLAobjectRoot* es la superclase de toda clase de objeto definida, es decir esta clase corresponde a la clase raíz de la jerarquía de descomposición de clases de objetos.

Los objetos pueden tener atributos que representan una parte del estado del objeto y cuyo valor puede variar en el tiempo. Los federados pueden suscribirse o publicar valores de atributos. El conocimiento de ciertas características de los atributos de un objeto, como ser su tipo de dato y las políticas de actualización forman parte de este contrato entre federados y son necesarios para permitir una eficiente comunicación entre federados durante la ejecución de la federación. Los atributos se definen con el tag <*attribute*> el cual tiene un conjunto de valores que deben ser especificados (representados como los atributos del elemento *attribute* en XML). Estos valores son: (i) *dataType* es el tipo de dato del atributo, y debe ser consistente con las definiciones de los tipos posibles que aparecen en el mismo FOM con el tag <*DataTypes*>; (ii) *updateType* son las políticas de actualización del atributo cuyos posibles valores son: *static, periodic, conditional* y *na* estos valores identifican que el valor del atributo no cambia, cambia periódicamente o cambia dada una condición especificada en *updateCondition* respectivamente; (iii) *updateCondition* son las condiciones de actualización, como ser por ejemplo el período en que debe actualizarse el valor del atributo; (iv) *ownership* son las formas de tener posesión del atributo para poder modificarlo; los posibles valores son: *NoTransfer, DivestAcquire*; el primer valor indica que ese atributo no puede ser transferido durante la ejecución, el segundo valor indica que el atributo puede ser liberado por su poseedor y poseído por otro federado; (v) *sharing* son las formas de compartir el atributo indicando si puede ser publicado, subscripto, ambas cosas o ninguna; (vi) *dimensions* identifican las asociaciones del atributo con un conjunto de dimensiones

las cuales deben estar definidas en el mismo FOM dentro del tag <*dimensions*>; (vii) *transportation* son las formas en que el atributo será transportado, estas formas deben estar definidas en el mismo FOM con el tag <*Transportations*>; (viii) *order* es el orden de entrega a ser usado para el valor de este atributo, sus posibles valores son: *receive* y *timeStamp* indicando que va a ser ordenado de acuerdo al tiempo en que fue recibido o respecto al tiempo de su timeStamp respectivamente.

Una interacción se define como una acción explícita tomada por un federado que puede tener algún efecto o impacto sobre otros federados dentro de la ejecución de la federación. La estructura de las interacciones es definida jerárquicamente a través de las relaciones clase-subclase. Las interacciones son un determinante en la interoperabilidad de los simuladores, ya que la misma requiere de una consistencia en el tratamiento de las mismas. Una interacción se define a través del tag <*interactionClass*> seguido por el nombre de la interacción (atributo *name* del elemento *interactionClass* en el archivo XML) y la información de las capacidades de publicar o suscribir dicha interacción (atributo *sharing* del elemento *interactionClass* en el archivo XML), los posibles valores para este último son: *Publish, Subscribe, Neither,* y *PublishSubscribe*. La clase *HLAinteractionRoot* es la superclase de toda clase de interacción definida, es decir esta clase corresponde a la clase raíz de la jerarquía de descomposición de clases de interacciones.

Las interacciones pueden tener parámetros. Un parámetro corresponde a información útil que está asociada a una interacción. A diferencia de los atributos de objetos, los parámetros de las interacciones no pueden ser suscriptos o publicados independientemente de la interacción en sí, por lo tanto, los valores asociados a la dimensión, orden de entrega y transporte es especificado a nivel de interacción más que a nivel de parámetros. Cuando una interacción es generada por un federado, es entregada con todos sus parámetros asociados, no es posible seleccionar un subconjunto de estos a enviar con la

interacción. Los parámetros se definen con el tag <*parameter*> seguido por el nombre del mismo (atributo *name* del elemento *parameter* en el archivo XML) y el tipo de dato (atributo *dataType* del elemento *parameter* en el archivo XML).

El conjunto de clases de objetos e interacciones predefinidos en HLA conocido como *MOM (Management Object Model)* se emplean para acceder a la información sobre la ejecución de la federación, interiorizarse sobre las operaciones de incorporación de federados a la federación (joinFederate) y de las operaciones ejecutadas por el RTI, controlar las funciones del RTI, la ejecución de la federación y de federados individuales que pertenecen a la federación, etc. Las clases que forman parte del MOM son por ejemplo *HLAfederation, HLAmanager, HLArequest, HLArequestSynchronizationPoints*, entre otros. Se reserva el prefijo *HLA* para los nombre de los elementos que forman el MOM, este prefijo no puede ser usado por las definiciones hechas por el usuario. Un FOM debe incluir a las definiciones de MOM.

A continuación se presenta la definición completa del FOM que fue utilizado en el ejemplo del capítulo 7.

```
<?xml version=\"1.0\" ?>
 <!DOCTYPE objectModel (View Source for full doctype...)>
 <objectModel DTDversion=\"1516.2\" name=\"Example\" type="FOM" version="1.0"
   date= "2000-04-01">
 <objects>
    <objectClass name="HLAobjectRoot"
     sharing="Neither">

     <attribute name="HLAprivilegeToDeleteObject"
      dataType="NA"
      updateType="NA"
      updateCondition="NA"
      ownership="NoTransfer"
      sharing="Neither"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"/>

    <objectClass name="UserBaseClass"
       sharing="Neither" >

      <objectClass name="StateOrderSource"
       sharing="PublishSubscribe"
```

```
 semantics="Clase que representa los posibles estados de las órdenes de clientes
 en el nodo source">

  <attribute name="delayed"
   dataType="integer"
   updateType="Static"
   updateCondition="NA"
   ownership="NoTransfer"
   sharing="PublishSubscribe"
   dimensions="NA"
   transportation="HLAreliable"
   order="TimeStamp"
   semantics= "cantidad de órdenes demoradas en el nodo source"/>

  <attribute name="filled"
   dataType="integer"
   updateType="Static"
   updateCondition="NA"
   ownership="NoTransfer"
   sharing="PublishSubscribe"
   dimensions="NA"
   transportation="HLAreliable"
   order="TimeStamp"
   semantics="cantidad de órdenes completas en el nodo source"/>

  <attribute name="unfilled"
   dataType="integer"
   updateType="Static"
   updateCondition="NA"
   ownership="NoTransfer"
   sharing="PublishSubscribe"
   dimensions="NA"
   transportation="HLAreliable"
   order="TimeStamp"
   semantics="cantidad de órdenes entregadas incompletas en el nodo source" />

</objectClass>

<objectClass name="StateOrderMake"
 sharing="PublishSubscribe"
 semantics="Clase que representa los posibles estados de las órdenes de clientes
 en el nodo source">

  <attribute name="delayed"
   dataType="integer"
   updateType="Static"
   updateCondition="NA"
   ownership="NoTransfer"
   sharing="PublishSubscribe"
   dimensions="NA"
   transportation="HLAreliable"
```

```
      order="TimeStamp"
      semantics="cantidad de órdenes demoradas en el nodo make"/>

     <attribute name="filled"
      dataType="integer"
      updateType="Static"
      updateCondition="NA"
      ownership="NoTransfer"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="cantidad de órdenes completas en el
      nodo make"/>

     <attribute name="unfilled"
      dataType="integer"
      updateType="Static"
      updateCondition="NA"
      ownership="NoTransfer"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="cantidad de órdenes entregadas incompletas en el nodo make" />

   </objectClass>

</objectClass>

<objectClass name="HLAmanager"
 sharing="Neither"
 semantics="This object class is the root class of all MOM object classes">

   <objectClass name="HLAfederate"
    sharing="Publish"
    semantics="This object class shall contain RTI state variables relating
    to a joined federate.>

       <attribute name="HLAfederateHandle"
        dataType="HLAhandle"
        updateType="Static"
        updateCondition="NA"
        ownership="NoTransfer"
        sharing="Publish"
        dimensions="Federate"
        transportation="HLAreliable"
        order="Receive"
        semantics="Handle of the joined federate returned by a Join Federation
        Execution service invocation" />
```

```
<attribute name="HLAfederateType"
 dataType="HLAunicodeString"
 updateType="Static"
 updateCondition="NA"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Type of the joined federate specified by the joined federate
 when it joined the federation" />

<attribute name="HLAfederateHost"
 dataType="HLAunicodeString"
 updateType="Static"
 updateCondition="NA"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Host name of the computer on which the joined federate is executing" />

<attribute name="HLARTIversion"
 dataType="HLAunicodeString"
 updateType="Static"
 updateCondition="NA"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Version of the RTI software being used" />

<attribute name="HLAFDDID"
 dataType="HLAunicodeString"
 updateType="Static"
 updateCondition="NA"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Identifier associated with the FDD data used by the joined federate" />

<attribute name="HLAtimeConstrained"
 dataType="HLAboolean"
 updateType="Conditional"
 updateCondition="Service invocation"
 ownership="NoTransfer"
 sharing="Publish"
```

```
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Whether the time advance of the joined federate is constrained by
 other joined federates" />

<attribute name="HLAtimeRegulating"
 dataType="HLAboolean"
 updateType="Conditional"
 updateCondition="Service invocation"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Whether the joined federate influences the time advancement of other
 joined federates"/>

<attribute name="HLAasynchronousDelivery"
 dataType="HLAboolean"
 updateType="Conditional"
 updateCondition="Service invocation"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Whether the RTI shall deliver receive-order messages to
 the joined federate while the joined federate's time manager state is not Time
 Advancing (only matters if the joined federate is time-constrained)" />

<attribute name="HLAfederateState"
 dataType="HLAfederateState"
 updateType="Conditional"
 updateCondition="Service invocation"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="State of the joined federate" />

<attribute name="HLAtimeManagerState"
 dataType="HLAtimeState"
 updateType="Conditional"
 updateCondition="Service invocation"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
```

```
 semantics="State of the joined federate's time manager" />

<attribute name="HLAlogicalTime"
 dataType="HLAlogicalTime"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Joined federate's logical time. Initial value of this information
 is initial value of time of the Time Representation Abstract datatype." />

<attribute name="HLAlookahead"
 dataType="HLAtimeInterval"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Minimum duration into the future that a TSO message will
 be scheduled. The value shall not be defined if the joined federate is not
 time-regulating)" />

<attribute name="HLAGALT"
 dataType="HLAlogicalTime"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Joined federate's Greatest Available Logical Time (GALT). The value
 shall not be defined if GALT is not defined for the joined federate." />

<attribute name="HLALITS"
 dataType="HLAlogicalTime"
 updateType="Periodic"
 updateCondition= "HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Joined federate's Least Incoming Time Stamp (LITS). The value shall
 not be defined if LITS is not defined for the joined federate." />
```

```
<attribute name="HLAROlength"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Number of RO messages queued for delivery to the joined federate." />

<attribute name="HLATSOlength"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Number of TSO messages queued for delivery to the joined federate" />

<attribute name="HLAreflectionsReceived"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of reflections received by the joined federate." />

<attribute name="HLAupdatesSent"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of updates sent by the joined federate" />

<attribute name="HLAinteractionsReceived"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
```

```
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of interactions received by the joined federate." />

<attribute name="HLAinteractionsSent"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of interactions sent by the joined federate. This
 information shall reflect related DDM usage." />

<attribute name="HLAobjectsInstancesThatCanBeDeleted"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of object instances whose
 HLAprivilegeToDeleteObject attribute is owned by the joined federate" />

<attribute name="HLAobjectInstancesUpdated"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of object instances for which the joined federate
 has invoked the Update Attribute Values service." />

<attribute name="HLAobjectInstancesReflected"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of object instances for which the joined
 federate has had a Reflect Attribute Values service invocation." />
```

```
<attribute name="HLAobjectInstancesDeleted"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of times the Delete Object Instance service
 was invoked by the joined federate since the federate joined the
 federation" />

<attribute name="HLAobjectInstancesRemoved"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of times the Remove Object Instance service
 was invoked for the joined federate since the federate joined the
 federation."/>

<attribute name="HLAobjectInstancesRegistered"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of times the Register Object Instance or
 Register Object Instance with Region service was invoked by the joined federate
 since the federate joined the federation." />

<attribute name="HLAobjectInstancesDiscovered"
 dataType="HLAcount"
 updateType="Periodic"
 updateCondition="HLAsetTiming.HLAreportPeriod"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="Federate"
 transportation="HLAreliable"
 order="Receive"
 semantics="Total number of times the Discover Object Instance
 service was invoked for the joined federate since the federate joined
```

```
      federation"/>

     <attribute name="HLAtimeGrantedTime"
      dataType="HLAmsec"
      updateType="Periodic"
      updateCondition="HLAsetTiming.HLAreportPeriod"
      ownership="NoTransfer"
      sharing="Publish"
      dimensions="Federate"
      transportation="HLAreliable"
      order="Receive"
      semantics="Wall clock time duration that the federate has spent in the
      Time Granted state since the last update of this attribute." />

     <attribute name="HLAtimeAdvancingTime"
      dataType="HLAmsec"
      updateType="Periodic"
      updateCondition="HLAsetTiming.HLAreportPeriod"
      ownership="NoTransfer"
      sharing="Publish"
      dimensions="Federate"
      transportation="HLAreliable"
      order="Receive"
      semantics="Wall clock time duration that the federate has spent in the Time
      Advancing state since the last update of this attribute." />

</objectClass>

<objectClass name="HLAfederation"
 sharing="Publish"
 semantics="This object class shall contain RTI state variablesrelating to a
 federation execution.">

     <attribute name="HLAfederationName"
      dataType="HLAunicodeString"
      updateType="Static"
      updateCondition="NA"
      ownership="NoTransfer"
      sharing="Publish"
      dimensions="NA"
      transportation="HLAreliable"
      order="Receive"
      semantics="Name of the federation to which the joined federate belongs" />

     <attribute name="HLAfederatesinFederation"
      dataType="HLAhandleList"
      updateType="Conditional"
      updateCondition="Federate joins or resigns"
      ownership="NoTransfer"
      sharing="Publish"
      dimensions="NA"
```

```
 transportation="HLAreliable"
 order="Receive"
 semantics="Identifiers of joined federates that are joined to the federation" />

<attribute name="HLARTIversion"
 dataType="HLAunicodeString"
 updateType="Static"
 updateCondition="NA"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="NA"
 transportation="HLAreliable"
 order="Receive"
 semantics="Version of RTI software"/>

<attribute name="HLAFDDID"
 dataType="HLAunicodeString"
 updateType="Static"
 updateCondition="NA"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="NA"
 transportation="HLAreliable"
 order="Receive"
 semantics="Identifier associated with the FDD used in the relevant Create
 Federation Execution service invocation." />

<attribute name="HLAlastSaveName"
 dataType="HLAunicodeString"
 updateType="Conditional"
 updateCondition="Service invocation"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="NA"
 transportation="HLAreliable"
 order="Receive"
 semantics="Name associated with the last federation state save" />

<attribute name="HLAlastSaveTime"
 dataType="HLAlogicalTime"
 updateType="Conditional"
 updateCondition="Service invocation"
 ownership="NoTransfer"
 sharing="Publish"
 dimensions="NA"
 transportation="HLAreliable"
 order="Receive"
 semantics="Logical time at which the last federation state timed save
 occurred." />

<attribute name="HLAnextSaveName"
```

```
          dataType="HLAunicodeString"
          updateType="Conditional"
          updateCondition="Service invocation"
          ownership="NoTransfer"
          sharing="Publish"
          dimensions="NA"
          transportation="HLAreliable"
          order="Receive"
          semantics="Name  associated with the next federation state save" />

        <attribute name="HLAnextSaveTime"
          dataType="HLAlogicalTime"
          updateType="Conditional"
          updateCondition="Service invocation"
          ownership="NoTransfer"
          sharing="Publish"
          dimensions="NA"
          transportation="HLAreliable"
          order="Receive"
          semantics="Logical time at which the next federation state timed save is
          scheduled." />

        <attribute name="HLAautoProvide"
          dataType="HLAswitch"
          updateType="Conditional"
          updateCondition="MOM interaction"
          ownership="NoTransfer"
          sharing="Publish"
          dimensions="NA"
          transportation="HLAreliable"
          order="Receive"
          semantics="Value of federation-wide Auto Provide Switch. Updated when value
          of switch changes" />

        <attribute name="HLAconveyRegionDesignatorSets"
          dataType="HLAswitch"
          updateType="Conditional"
          updateCondition="MOM interaction"
          ownership="NoTransfer"
          sharing="Publish"
          dimensions="NA"
          transportation="HLAreliable"
          order="Receive"
          semantics="Value of federation-wide Convey Region Designator Sets Switch.
          Updated when value of switch changes" />

      </objectClass>

    </objectClass>

  </objectClass>
```

```
</objects>

<interactions>

   <interactionClass name="HLAinteractionRoot"
     sharing="Neither"
     dimensions="NA"
     transportation="HLAreliable"
     order="Receive">

       <interactionClass name="UserInteractionBase"
         sharing="Neither"
         dimensions="NA"
         transportation="HLAreliable"
         order="TimeStamp"
         semantics="Base classs of user-defined interactions">

           <interactionClass name="Ask"
             sharing="PublishSubscribe"
             dimensions="NA"
             transportation="HLAreliable"
             order="TimeStamp"
             semantics="SubClass of UserInteractionBase">

               <interactionClass name="AskPaiment"
                 sharing="PublishSubscribe"
                 dimensions="NA"
                 transportation="HLAreliable"
                 order="TimeStamp"
                 semantics="SubClass of ASK">

                 <parameter name="billNro"
                   dataType="HLAreliable"
                   semantics="nro de la factura a pagar" />

                 <parameter name="amount"
                   dataType="HLAreliable"
                   semantics="monto de la factura a pagar" />

               </interactionClass>

               <interactionClass name="AskGoods"
                 sharing="PublishSubscribe"
                 dimensions="NA"
                 transportation="HLAreliable"
                 order="TimeStamp"
                 semantics="SubClass of UserInteractionBase">

                  <parameter name="item"
                    dataType="HLAreliable"
```

```
                    semantics="código o nombre de la mercadería solicitada" />

                  <parameter name="customerOrderNro"
                    dataType="HLAreliable"
                    semantics="nro de la orden del cliente" />

                  <parameter name="quantity"
                    dataType="HLAreliable"
                    semantics="cantidad solicitada del item" />

              </interactionClass>
<interactionClass name="AskSupply"
                  sharing="PublishSubscribe"
                  dimensions="NA"
                  transportation="HLAreliable"
                  order="TimeStamp"
                  semantics="SubClass of UserInteractionBase">

                   <parameter name="item"
                     dataType="HLAreliable"
                     semantics="código o nombre de la mercadería solicitada" />

                   <parameter name="sourceOrderNro"
                     dataType="HLAreliable"
                     semantics="nro de la orden del distribuidor" />

                   <parameter name="quantity"
                     dataType="HLAreliable"
                     semantics="cantidad solicitada del item" />

                </interactionClass>

            </interactionClass>

            <interactionClass name="Complaint"
              sharing="PublishSubscribe"
              dimensions="NA"
              transportation="HLAreliable"
              order="TimeStamp"
              semantics="SubClass of UserInteractionBase">

                <interactionClass name="ComplaintPaiment"
                  sharing="PublishSubscribe"
                  dimensions="NA"
                  transportation="HLAreliable"
                  order="TimeStamp"
                  semantics="SubClass of Complaint">

                   <parameter name="date"
                     dataType="HLAreliable"
                     semantics="fecha del reclamo"/>
```

```
        <parameter name="amount"
          dataType="HLAreliable"
          semantics="monto reclamado"/>

        <parameter name="billReference"
          dataType="HLAreliable"
          semantics="Nro de la factura reclamada" />

    </interactionClass>

    <interactionClass name="ComplaintGoods"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="SubClass of Complaint">

        <parameter name="quantity"
          dataType="HLAreliable"
          semantics="cantidad reclamada del item" />

        <parameter name="item"
          dataType="HLAreliable"
          semantics="item reclamado" />

        <parameter name="customerOrderReference"
          dataType="HLAreliable"
          semantics="nro de la orden del cliente reclamada" />

    </interactionClass>

</interactionClass>

<interactionClass name="Deliver"
  sharing="PublishSubscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="TimeStamp"
  semantics="SubClass of UserInteractionBase">

    <interactionClass name="DeliverGoods"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="SubClass of Deliver">

        <parameter name="itemID"
          dataType="HLAreliable"
          semantics="identificador del item"/>
```

```
        <parameter name="date"
          dataType="HLAreliable"
          semantics="fecha de la entrega"/>

        <parameter name="quantity"
          dataType="HLAreliable"
          semantics="cantidad entregada del item" />

    </interactionClass>

</interactionClass>

<interactionClass name="Return"
  sharing="PublishSubscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="TimeStamp"
  semantics="SubClass of UserInteractionBase">

    <interactionClass name="ReturnBadGoods"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="SubClass of Return">

       <parameter name="itemID"
         dataType="HLAreliable"
         semantics="identificador del item devuelto" />

       <parameter name="customerOrderReference"
         dataType="HLAreliable"
         semantics="referencia a la orden del cliente" />

       <parameter name="quantity"
         dataType="HLAreliable"
         semantics="cantidad devuelta del item" />

    </interactionClass>

    <interactionClass name="ReturnExcecedGoods"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="SubClass of Return">

       <parameter name="itemID"
         dataType="HLAreliable"
         semantics="identificador del item devuelto" />
```

```
        <parameter name="customerOrderReference"
          dataType="HLAreliable"
          semantics="referencia a la orden del cliente" />

        <parameter name="quantity"
          dataType="HLAreliable"
          semantics="cantidad solicitada del item" />

    </interactionClass>

</interactionClass>

<interactionClass name="Send"
  sharing="PublishSubscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="TimeStamp"
  semantics="SubClass of UserInteractionBase">

    <interactionClass name="SendPayment"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="SubClass of Send">

       <parameter name="customerOrderID"
         dataType="HLAreliable"
         semantics="identificador de la orden pagada" />

       <parameter name="amount"
         dataType="HLAreliable"
         semantics="cantidad de dinero enviado"/>

    </interactionClass>

    <interactionClass name="SendDocument"
      sharing="PublishSubscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="TimeStamp"
      semantics="SubClass of Send">

      <parameter name="type"
        dataType="HLAreliable"
        semantics="document type sending (bill, customer order, etc)" />

      <parameter name="documentID"
        dataType="HLAreliable"
        semantics="identificador del documento enviado" />
```

```
          </interactionClass>

      </interactionClass>

</interactionClass>

<interactionClass name="HLAmanager"
  sharing="Neither"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Root class of MOM interactions">

   <interactionClass name="HLAfederate"
     sharing="Neither"
     dimensions="NA"
     transportation="HLAreliable"
     order="Receive"
     semantics="Root class of MOM interactions that deal with a specific
     joined federate">

     <parameter name="HLAfederate"
       dataType="HLAhandle"
       semantics="Handle of the joined federate that was provided when
       joining." />

    <interactionClass name="HLAadjust"
      sharing="Neither"
      dimensions="NA"
      transportation="HLAreliable"
      order="Receive"
      semantics="Permit a joined federate to adjust the RTI statevariables
      associated with another joined federate">

        <interactionClass name="HLAsetTiming"
          sharing="Subscribe"
          dimensions="NA"
          transportation="HLAreliable"
          order="Receive"
          semantics="Adjust the time period updates of the
          HLAmanager.HLAfederate object instance for thespecified
          joined federate.">

           <parameter name="HLAreportPeriod"
             dataType="HLAseconds"
             semantics="Number of seconds between updates of instance
             attribute values of the HLAfederate object instance" />

       </interactionClass>
```

```
<interactionClass name="HLAmodifyAttributeState"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Modify the ownership state of an attribute of an object
  instance for the specified joined federate.">

   <parameter name="HLAobjectInstance"
     dataType="HLAhandle"
     semantics="Handle of the object instance whose attribute state
     is being changed" />

   <parameter name="HLAattribute"
     dataType="HLAhandle"
     semantics="Handle of the instance attribute whose stateis
     being changed" />

   <parameter name="HLAattributeState"
     dataType="HLAownership"
     semantics="New state for the attribute of the object Instance"/>

</interactionClass>

<interactionClass name="HLAsetServiceReporting"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Specify whether to report service invocations to0 or
  from the specified joined federate via
  HLAmanager.HLAfederate. HLAreport.HLAreportServiceInvocation
  interactions (enable or disable servicereporting).">

   <parameter name="HLAreportingState"
     dataType="HLAboolean"
     semantics="Whether the RTI should report service invocations" />

</interactionClass>

<interactionClass name="HLAsetExceptionReporting"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Specify whether the RTI shall report service invocation
  exceptions via HLAmanager.HLAfederate.HLAreport. HLAreportException
  interactions">

   <parameter name="HLAreportingState"
     dataType="HLAboolean"
```

```
            semantics="Whether the RTI should report exceptions" />

      </interactionClass>

   </interactionClass>

   <interactionClass name="HLArequest"
     sharing="Neither"
     dimensions="NA"
     transportation="HLAreliable"
     order="Receive"
     semantics="Permit a federate to request RTI data about another federate">

      <interactionClass name="HLArequestPublications"
        sharing="Subscribe"
        dimensions="NA"
        transportation="HLAreliable"
        order="Receive"
        semantics="Request that the RTI send report interactions that contain the
        publication data of a joined federate." />

      <interactionClass name="HLArequestSubscriptions"
        sharing="Subscribe"
        dimensions="NA"
        transportation="HLAreliable"
        order="Receive"
        semantics="Request that the RTI send report interactions that contain
        the subscription data of a joined federate." />

      <interactionClass name="HLArequestObjectInstancesThatCanBeDeleted"
        sharing="Subscribe"
        dimensions="NA"
        transportation="HLAreliable"
        order="Receive"
        semantics="Request that the RTI send a report interaction that
        contains the object instances that can be deleted at the joined
         federate." />

      <interactionClass name="HLArequestObjectInstancesUpdated"
        sharing="Subscribe"
        dimensions="NA"
        transportation="HLAreliable"
        order="Receive"
        semantics="Request that the RTI send a report interaction that
        contains the object instance updating responsibility of a
        joined federate." />

      <interactionClass name="HLArequestObjectInstancesReflected"
        sharing="Subscribe"
        dimensions="NA"
        transportation="HLAreliable"
```

```
  order="Receive"
  semantics="Request that the RTI send a report interaction that contains
  the objects instances for which a joined federate has had a Reflect
  Attribute Values service invocation." />

<interactionClass name="HLArequestUpdatesSent"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request that the RTI send a report interaction that contains
  the number of updates that a joined federate has generated." />

<interactionClass name="HLArequestInteractionsSent"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request that the RTI send a report interaction that
  contains the number of interactions that a joined federate has generated."/>

<interactionClass name="HLArequestReflectionsReceived"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request that the RTI send a report interaction that
  contains the number of reflections that a joined federate has
  received." />

<interactionClass name="HLArequestInteractionsReceived"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request that the RTI send a report interaction that
  contains the number of interactions that a joined federate has
  received." />

<interactionClass name="HLArequestObjectInstanceInformation"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request that the RTI send a report interaction that contains
  the information that a joined federate maintains on a single object
  instance.">

   <parameter name="HLAobjectInstance"
     dataType="HLAhandle"
     semantics="Handle of the object instance for which information is
```

```
          being requested" />

    </interactionClass>

    <interactionClass name="HLArequestSynchronizationPoints"
      sharing="Subscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="Receive"
      semantics="Request that the RTI send a report interaction that contains
      a list of all in-progress federation synchronization points." />

    <interactionClass name="HLArequestSynchronizationPointStatus"
      sharing="Subscribe"
      dimensions="NA"
      transportation="HLAreliable"
      order="Receive"
      semantics="Request that the RTI send a report interaction that contains
      a list that includes each federate that is associated with a particular
      synchronization point." />

</interactionClass>

<interactionClass name="HLAreport"
  sharing="Neither"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Report RTI data about a joined federate.">

<interactionClass name="HLAreportObjectClassPublication"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response to an
  interaction of class
  HLAmanager.HLAfederate.HLArequest.HLArequestPublications.">

  <parameter name="HLAnumberOfClasses"
    dataType="HLAcount"
    semantics="The number of object classes for which the joined federate
    publishes attributes" />

  <parameter name="HLAobjectClass"
   dataType="HLAhandle"
   semantics="The object class whose publication is being reported" />

  <parameter name="HLAattributeList"
    dataType="HLAhandleList"
    semantics="List of handles of HLAobjectClass attributes that the
```

```
    joined federate is publishing" />

</interactionClass>

<interactionClass name="HLAreportObjectClassSubscription"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response to an
  interaction of class
  HLAmanager.HLAfederate.HLArequest.HLArequestSubscriptions.">

   <parameter name="HLAnumberOfClasses"
     dataType="HLAcount"
     semantics="The number of object classes for which the joined
     federate subscribes to attributes." />

   <parameter name="HLAobjectClass"
     dataType="HLAhandle"
     semantics="The object class whose subscription is being reported" />

   <parameter name="HLAactive"
     dataType="HLAboolean"
     semantics="Whether the subscription is active" />

   <parameter name="HLAattributeList"
     dataType="HLAhandleList"
     semantics="List of handles of class attributes to which the joined
     federate is subscribing." />

</interactionClass>

<interactionClass name="HLAreportInteractionPublication"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response to
  an interaction of class
  HLAmanager.HLAfederate. HLArequest.HLArequestPublications.">

   <parameter name="HLAinteractionClassList"
     dataType="HLAhandleList" semantics="List
     of interaction classes that the joined federate is publishing" />

</interactionClass>

<interactionClass name="HLAreportInteractionSubscription"
  sharing="Publish"
  dimensions="Federate"
```

```
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response to
  an interaction of class
  HLAmanager.HLAfederate.HLArequest.HLArequestSubscriptions.">

   <parameter name="HLAinteractionClassList"
     dataType="HLAinteractionSubList"
     semantics="List of interaction class/subscription type pairs." />

</interactionClass>

<interactionClass name="HLAreportObjectInstancesThatCanBeDeleted"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response to
  an interaction of class
  HLAmanager.HLAfederate.HLArequest.HLArequestObject
  InstancesThatCanBeDeleted">

   <parameter name="HLAobjectInstanceCounts"
     dataType="HLAobjectClassBasedCounts"
     semantics="A list of object instance counts." />

</interactionClass>

<interactionClass name="HLAreportObjectInstancesUpdated"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response
  to an interaction of class
  HLAmanager.HLAfederate.HLArequest.HLArequestObjectInstancessUpdated.">

   <parameter name="HLAobjectInstanceCounts"
     dataType="HLAobjectClassBasedCounts"
     semantics="List of object instance counts. Each object instance
     count consists of an object class handle and the number of object
     instances of that class." />

</interactionClass>

<interactionClass name="HLAreportObjectInstancesReflected"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI inresponse to
```

```
  an interaction of class
  HLAmanager.HLAfederate.HLArequest.HLArequestObjectInstancessReflected">

  <parameter name="HLAobjectInstanceCounts"
    dataType="HLAobjectClassBasedCounts"
    semantics="List of object instance counts. Each object instance
    count consists of an object class handle and the number of object
    instances of that class."/>

</interactionClass>

<interactionClass name="HLAreportUpdatesSent"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response to
  an interaction of class
  HLAmanager.HLAfederate. HLArequest.HLArequestUpdatesSent.">

  <parameter name="HLAtransportation"
    dataType="HLAtransportationName"
    semantics="Transportation type used in sending updates" />

  <parameter name="HLAupdateCounts"
    dataType="HLAobjectClassBasedCounts"
    semantics="List of update counts. Each update count consists of
    an object class handle and the number of updates sent of that class" />

</interactionClass>

<interactionClass name="HLAreportReflectionsReceived"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response to an
  interaction class
  HLAmanager.HLAfederate.HLArequest.HLArequestReflectionsReceived">

  <parameter name="HLAtransportation"
    dataType="HLAtransportationName"
    semantics="Transportation type used in receiving reflections" />

  <parameter name="HLAreflectCounts"
    dataType="HLAobjectClassBasedCounts"
    semantics="List of reflection counts. Each reflection count
    consists of an object class handle and the number of reflections
    received of that class."/>

</interactionClass>
```

```
<interactionClass name="HLAreportInteractionsSent"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response
  to an interaction of class
  HLAmanager.HLAfederate.HLArequest.HLArequestInteractionsSent.">

  <parameter name="HLAtransportation"
    dataType="HLAtransportationName"
    semantics="Transportation type used in sending interactions" />

  <parameter name="HLAinteractionCounts"
    dataType="HLAinteractionCounts"
    semantics="List of interaction counts. Each interaction count
    consists of an interaction class handle and the number of
    interactions of that class." />

</interactionClass>

<interactionClass name="HLAreportInteractionsReceived"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response
  to an interaction of class
  HLAmanager.HLAfederate.HLArequest. HLArequestInteractionsReceived">

  <parameter name="HLAtransportation"
    dataType="HLAtransportationName"
    semantics="Transportation type used in receiving interactions" />

  <parameter name="HLAinteractionCounts"
    dataType="HLAinteractionCounts"
    semantics="List of interaction counts. Each interaction count
    consists of an interaction class handle and the number of
    interactions of that class." />

</interactionClass>

<interactionClass name="HLAreportObjectInstanceInformation"
  sharing="Publish"
  dimensions="Federate"
  transportation="HLAreliable"
  order="Receive"
  semantics="The interaction shall be sent by the RTI in response to
  an interaction of class
  HLAmanager.HLAfederate.HLArequest.HLArequestObjectInstance Information">
```

```
    <parameter name="HLAobjectInstance"
      dataType="HLAhandle"
      semantics="Handle of the object instance for which the interaction
      was sent" />

    <parameter name="HLAownedInstanceAttributeList"
      dataType="HLAhandleList"
      semantics="List of the handles of all instance attributes, of the
      object instance, owned by the joined federate" />

    <parameter name="HLAregisteredClass"
      dataType="HLAhandle"
      semantics="Handle of the registered class of the object instance" />

    <parameter name="HLAknownClass"
      dataType="HLAhandle"
      semantics="Handle of the known class of the object instance at the
      joined federate" />

  </interactionClass>

  <interactionClass name="HLAreportException"
    sharing="Publish"
    dimensions="Federate"
    transportation="HLAreliable"
    order="Receive"
    semantics="The interaction shall be sent by the RTI when an exception
    occurs as the result of a service invocation at the indicated
    joined federate.">

    <parameter name="HLAservice"
      dataType="HLAunicodeString"
      semantics="Name of the service that raised the exception" />

    <parameter name="HLAexception"
      dataType="HLAunicodeString"
      semantics="Textual depiction of the exception" />

  </interactionClass>

  <interactionClass name="HLAreportServiceInvocation"
    sharing="Publish"
    dimensions="Federate ServiceGroup"
    transportation="HLAreliable"
    order="Receive"
    semantics="This interaction shall be sent by the RTI whenever an HLA
    service is invoked, either by the indicated joined federate or by the
    RTI at the indicated joined federate, and Service Reporting is Enabled
    for the indicated joined federate.">
```

```
    <parameter name="HLAservice"
      dataType="HLAunicodeString"
      semantics="Textual name of the service" />

    <parameter name="HLAsuccessIndicator"
      dataType="HLAboolean"
      semantics="Whether the service invocation was successful.Exception
      values are returned along with HLAfalse value" />

    <parameter name="HLAsuppliedArguments"
      dataType="HLAargumentList"
      semantics="Textual depiction of the arguments supplied in the service
     invocation"/>

    <parameter name="HLAreturnedArguments"
       dataType="HLAargumentList"
       semantics="Textual depiction of the argument returned by the service
       invocation" />

    <parameter name="HLAexception"
      dataType="HLAunicodeString"
      semantics="Textual
      depiction of the exception raised by this service invocation" />

    <parameter name="HLAserialNumber"
      dataType="HLAcount"
      semantics="This is a per-joined federate serial number that shall
      start at zero and shall increment by 1 for each
      HLAmanager.HLAfederate.HLAreport HLAreportServiceInvocation interaction
      that represents service invocations to or from the respective joined
      federate." />

  </interactionClass>

  <interactionClass name="HLAreportMOMexception"
    sharing="Publish"
    dimensions="Federate"
    transportation="HLAreliable"
    order="Receive"
    semantics="The interaction shall be sent by the RTI when one the following
    occurs:- a MOM interaction without all the necessary parameters is sent
    or - an interaction that imitates a federate's invocation of an RTI service
    is sent and not all of the service's pre-conditions are met.">

    <parameter name="HLAservice"
      dataType="HLAunicodeString"
      semantics="Name of the service interaction that had a problem or raised
      the exception" />

    <parameter name="HLAexception"
      dataType="HLAunicodeString"
```

```
        semantics="Textual depiction of the problem or exception" />

      <parameter name="HLAparameterError"
        dataType="HLAboolean"
        semantics="HLAtrue if there was an incorrect number of interaction
        parameters, HLAfalse otherwise"/>

    </interactionClass>

    <interactionClass name="HLAreportSynchronizationPoints"
      sharing="Publish"
      dimensions="Federate"
      transportation="HLAreliable"
      order="Receive"
      semantics="The interaction shall be sent by the RTI in response to an
      interaction of class
      HLAmanager.HLAfederate.HLArequest.HLArequestSynchronizationPoints.">

      <parameter name="HLAsynchPoints"
        dataType="HLAsynchPointList"
        semantics="List of the in progress federation execution synchronization
        points" />

    </interactionClass>

    <interactionClass name="HLAreportSynchronizationPointStatus"
      sharing="Publish"
      dimensions="Federate"
      transportation="HLAreliable"
      order="Receive"
      semantics="The interaction shall be sent by the RTI in response to an
      interaction of class
      HLAmanager.HLAfederate. HLArequest.HLArequestSynchronizationPointStatus.">

      <parameter name="HLAsynchPointName"
        dataType="HLAunicodeString"
        semantics="Name of a particular synchronization point" />

      <parameter name="HLAsynchPointFederates"
        dataType="HLAsynchPointFederateList"
        semantics="List of each federate associated
        with the particular synchronization point" />

    </interactionClass>

  </interactionClass>

  <interactionClass name="HLAservice"
    sharing="Neither"
    dimensions="NA"
    transportation="HLAreliable"
```

```
order="Receive"
semantics="The interaction class shall be acted upon by the RTI.">

<interactionClass name="HLAresignFederationExecution"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Cause the joined federate to resign from the federation
  execution. A joined federate shall be able to send this interaction
  anytime.">

  <parameter name="HLAresignAction"
    dataType="HLAresignAction"
    semantics="Action that the RTI is to take in conjunction with the
    resignation"/>

</interactionClass>

<interactionClass name="HLAsynchronizationPointAchieved"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Mimic the federate's report of achieving a synchronization
  point.">

  <parameter name="HLAlabel"
    dataType="HLAunicodeString"
    semantics="Label associated with the synchronization point" />

</interactionClass>

<interactionClass name="HLAfederateSaveBegun"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Mimic the federate's report of starting a save" />

<interactionClass name="HLAfederateSaveComplete"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Mimic the joined federate's report of completion of a save.
  A joined federate shall be able to send this interaction during a
  federation save.">

  <parameter name="HLAsuccessIndicator"
    dataType="HLAboolean"
```

```
    semantics="Whether the save was successful" />

</interactionClass>

<interactionClass name="HLAfederateRestoreComplete"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Mimic the joined federate's report of completion of a restore.
  A joined federate shall be able to send this interaction during a
  federation restore.">

  <parameter name="HLAsuccessIndicator"
    dataType="HLAboolean"
    semantics="Whether the restore was successful" />

</interactionClass>

<interactionClass name="HLApublishObjectClassAttributes"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Set the joined federate's publication status of attributes
  belonging to an object class">

  <parameter name="HLAobjectClass"
    dataType="HLAhandle"
    semantics="Object class for which the joined federate's publication
    shall change" />

  <parameter name="HLAattributeList"
    dataType="HLAhandleList"
    semantics="List of handles of attributes of HLAobjectClass, which
    the federate shall now publish"/>

</interactionClass>

<interactionClass name="HLAunpublishObjectClassAttributes"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Cause the joined federate no longer to publish attributes
  of an object class">

  <parameter name="HLAobjectClass"
    dataType="HLAhandle"
    semantics="Object class for which the joined federate's unpublication
    shall change" />
```

```
    <parameter name="HLAattributeList"
      dataType="HLAhandleList"
      semantics="List of handles of attributes of HLAobjectClass, which the
      joined federate shall now unpublish" />

  </interactionClass>

  <interactionClass name="HLApublishInteractionClass"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Set the joined federate's publication status of an interaction
    class">

    <parameter name="HLAinteractionClass"
      dataType="HLAhandle"
      semantics="Interaction class that the joined federate shall publish" />

  </interactionClass>

  <interactionClass name="HLAunpublishInteractionClass"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Cause the joined federate no longer to publish an interaction
    class">

    <parameter name="HLAinteractionClass"
      dataType="HLAhandle"
      semantics="Interaction class that the joined federate shall no longer
      publish" />

  </interactionClass>

  <interactionClass name="HLAsubscribeObjectClassAttributes"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Set the joined federate's subscription status of attributes
    belonging to an object class">

    <parameter name="HLAobjectClass"
      dataType="HLAhandle"
      semantics="Object class for which the joined federate's subscription
      shall change" />

    <parameter name="HLAattributeList"
```

```
    dataType="HLAhandleList"
    semantics="List of handles of attributes of HLAobjectClass to which the
    joined federate shall now subscribe" />

  <parameter name="HLAactive"
    dataType="HLAboolean"
    semantics="Whether the subscription is active" />

</interactionClass>

<interactionClass name="HLAunsubscribeObjectClassAttributes"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Cause the joined federate no longer to subscribe to attributes
  of an object class">

  <parameter name="HLAobjectClass"
    dataType="HLAhandle"
    semantics="Object class for which the joined federate's subscription
    shall change"/>

  <parameter name="HLAattributeList"
    dataType="HLAhandleList"
    semantics="List of handles of attributes of HLAobjectClass to which
    the joined federate shall now unsubscribe" />

</interactionClass>

<interactionClass name="HLAsubscribeInteractionClass"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Set the joined federate's subscription status to an interaction
  class">

  <parameter name="HLAinteractionClass"
    dataType="HLAhandle"
    semantics="Interaction class to which the federate shall subscribe" />

  <parameter name="HLAactive"
    dataType="HLAboolean"
    semantics="Whether the subscription is active" />

</interactionClass>

<interactionClass name="HLAunsubscribeInteractionClass"
  sharing="Subscribe"
  dimensions="NA" transportation="HLAreliable" order="Receive"
```

```
    semantics="Cause the joined federate no longer to subscribe to an
    interaction class">

    <parameter name="HLAinteractionClass"
      dataType="HLAhandle"
      semantics="Interaction class to which the joined federate will no
      longer be subscribed" />

  </interactionClass>

  <interactionClass name="HLAdeleteObjectInstance"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Cause an object instance to be deleted from the federation.">

    <parameter name="HLAobjectInstance"
      dataType="HLAhandle"
      semantics="Handle of the object instance that is to be deleted" />

    <parameter name="HLAtag" dataType="HLAopaqueData"
     semantics="Tag associated with the deletion" />

    <parameter name="HLAtimeStamp"
      dataType="HLAlogicalTime"
      semantics="Time stamp of the deletion (optional)" />

  </interactionClass>

  <interactionClass name="HLAlocalDeleteObjectInstance"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Inform the RTI that it shall treat the specified object
    instance as if the joined federate did not know about the object
    instance.">

    <parameter name="HLAobjectInstance"
      dataType="HLAhandle"
      semantics="Handle of the object instance that is to be deleted" />

  </interactionClass>

  <interactionClass name="HLAchangeAttributeTransportationType"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Change the transportation type used by the joined federate
```

```
  when sending attributes belonging to the object instance">

  <parameter name="HLAobjectInstance"
    dataType="HLAhandle"
    semantics="Handle of the object instance whose attribute transportation
    type is to be changed" />

  <parameter name="HLAattributeList"
    dataType="HLAhandleList"
    semantics="List of the handles of instance attributes whose transportation
    type is to be changed" />

  <parameter name="HLAtransportation"
    dataType="HLAtransportationName"
    semantics="Transportation type to be used for updatinginstance attributes
    in the list" />

</interactionClass>

<interactionClass name="HLAchangeInteractionTransportationType"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Change the transportation type used by the joined federate when
  sending a class of interaction">

  <parameter name="HLAinteractionClass"
    dataType="HLAhandle"
    semantics="Interaction class whose transportation type is changed by this
    service invocation" />

  <parameter name="HLAtransportation"
    dataType="HLAtransportationName"
    semantics="Transportation type to be used for sending the interaction
    class"/>

</interactionClass>

<interactionClass name="HLAunconditionalAttributeOwnershipDivestiture"
   sharing="Subscribe"
   dimensions="NA"
   transportation="HLAreliable"
   order="Receive"
   semantics="Cause the ownership of attributes of an object instance to be
   unconditionally divested by the joined federate">

  <parameter name="HLAobjectInstance"
    dataType="HLAhandle"
    semantics="Handle of the object instance whose attributes'ownership is to
    be divested" />
```

```
    <parameter name="HLAattributeList"
      dataType="HLAhandleList"
      semantics="List of handles of instance attributes belonging to
      HLAobjectInstance whose ownership is to be divested by the joined federate"/>

  </interactionClass>

  <interactionClass name="HLAenableTimeRegulation"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Cause the joined federate to begin regulating the logical time
    of other joined federates">

    <parameter name="HLAlookahead"
      dataType="HLAtimeInterval"
      semantics="Lookahead to be used by the joined federate while regulating
      other joined federates" />

  </interactionClass>

  <interactionClass name="HLAdisableTimeRegulation"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Cause the joined federate to cease regulating the logical time of
    other joined federates" />

  <interactionClass name="HLAenableTimeConstrained"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Cause the logical time of the joined federate to begin being
    constrained by the logical times of other joined federates" />

  <interactionClass name="HLAdisableTimeConstrained"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
    order="Receive"
    semantics="Cause the logical time of the joined federate to cease being
    constrained by the logical times of other joined federates" />

  <interactionClass name="HLAtimeAdvanceRequest"
    sharing="Subscribe"
    dimensions="NA"
    transportation="HLAreliable"
```

```
  order="Receive"
  semantics="Request an advance of the joined federate's logical time on
  behalf of the joined federate, and release zero or more messages for
  delivery to the joined federate">

  <parameter name="HLAtimeStamp"
    dataType="HLAlogicalTime"
    semantics="Time stamp requested" />

</interactionClass>

<interactionClass name="HLAtimeAdvanceRequestAvailable"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request an advance of the joined federate's logical time, on
  behalf of the joined federate, and release zero or more messages for
  delivery to the joined federate">

  <parameter name="HLAtimeStamp"
    dataType="HLAlogicalTime"
    semantics="Time stamp requested" />

</interactionClass>

<interactionClass name="HLAnextMessageRequest"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request the logical time of the joined federate to be advanced
  to the time stamp of the next TSO message.">

  <parameter name="HLAtimeStamp"
    dataType="HLAlogicalTime"
    semantics="Time stamp requested" />

</interactionClass>

<interactionClass name="HLAnextMessageRequestAvailable"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request the logical time of the joined federate to be advanced
  to the time stamp of the next TSO message.">

  <parameter name="HLAtimeStamp"
    dataType="HLAlogicalTime"
    semantics="Time stamp requested" />
```

```
</interactionClass>

<interactionClass name="HLAflushQueueRequest"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Request the logical time of the joined federate to be advanced
  as far as possible, provided that the time stamp is less than or equal
  to the logical time specified in the request.">

  <parameter name="HLAtimeStamp"
    dataType="HLAlogicalTime"
    semantics="Time stamp requested" />

</interactionClass>

<interactionClass name="HLAenableAsynchronousDelivery"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Cause the RTI to deliver receive-order messages to the joined
  federate at any time, even if the joined federate is time-constrained." />

<interactionClass name="HLAdisableAsyncrhonousDelivery"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="When the joined federate is time-constrained, cause the RTI
  to deliver receive-order messages to the joined federate only when its
  time manager state is Time Advancing." />

<interactionClass name="HLAmodifyLookahead"
  sharing="Subscribe"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Change the lookahead value used by the joined federate">

  <parameter name="HLAlookahead"
    dataType="HLAtimeInterval"
    semantics="New value for lookahead" />

</interactionClass>

<interactionClass name="HLAchangeAttributeOrderType"
  sharing="Subscribe"
  dimensions="NA"
```

```
          transportation="HLAreliable"
          order="Receive"
          semantics="Change the ordering type used by the joined federate when
          sending attributes belonging to the object instance">

          <parameter name="HLAobjectInstance"
            dataType="HLAhandle"
            semantics="Handle of the object instance whose attribute order type
            is to be changed"/>

          <parameter name="HLAattributeList"
            dataType="HLAhandleList"
            semantics="List of the handles of instance attributes whose order type
            is to be changed" />

          <parameter name="HLAsendOrder"
            dataType="HLAorderType"
            semantics="Order type to be used for sending the instance attribute list" />

        </interactionClass>

        <interactionClass name="HLAchangeInteractionOrderType"
          sharing="Subscribe"
          dimensions="NA"
          transportation="HLAreliable"
          order="Receive"
          semantics="Change the order type used by the joined federate when sending a
          class of interaction">

          <parameter name="HLAinteractionClass"
            dataType="HLAhandle"
            semantics="Interaction class whose order type is changed by this
            service invocation"/>

          <parameter name="HLAsendOrder"
            dataType="HLAorderType"
            semantics="Order type to be used for sending the interaction class" />

        </interactionClass>

      </interactionClass>

</interactionClass>

<interactionClass name="HLAfederation"
  sharing="Neither"
  dimensions="NA"
  transportation="HLAreliable"
  order="Receive"
  semantics="Root class of MOM interactions that deal with a specific federation
  execution.">
```

```
      <interactionClass name="HLAadjust"
        sharing="Neither"
        dimensions="NA"
        transportation="HLAreliable"
        order="Receive"
        semantics="Permit a federate to adjust the RTI state variables associated
        with a federation execution.">

          <interactionClass name="HLAsetSwitches"
            sharing="Subscribe"
            dimensions="NA"
            transportation="HLAreliable" order="Receive" semantics="Set the values of
            several HLA switches.">

            <parameter name="HLAautoProvide"
              dataType="HLAswitch"
              semantics="Set the federation-wide Auto Provide Switch to the provided
              value (this parameter is required only when a change of this switch
              value is needed)." />

            <parameter name="HLAconveyRegionDesignatorSets"
              dataType="HLAswitch"
              semantics="Set the federation-wide Convey Region Designator Sets Switch
              to the provided value (this parameter is required only when a change
              of this switch value is needed)." />

          </interactionClass>

      </interactionClass>

    </interactionClass>

  </interactionClass>

 </interactionClass>

</interactions>

<dimensions>

  <dimension name="Federate"
    dataType="HLAfederateHandle"
    upperBoundNotes="MOM1"
    normalization="Normalize Federate Handle service"
    value="Excluded" />

  <dimension name="ServiceGroup"
    dataType="HLAserviceGroupName"
    upperBound="7"
    normalization="Normalize Service Group service"
```

```
    value="Excluded" />

</dimensions>

<time>

  <timeStamp dataType="NA" />

  <lookahead dataType="NA" />

</time>

<transportations>

  <transportation name="HLAreliable"
    description="Provide reliable delivery of data in the
    sense that TCP/IP delivers its data reliably" />

  <transportation name="HLAbestEffort"
    description="Make an effort to deliver data in the
    sense that UDP provides best-effort delivery" />

</transportations>

<switches autoProvide="Enabled"
  conveyRegionDesignatorSets="Enabled"
  attributeScopeAdvisory="Enabled"
  attributeRelevanceAdvisory="Enabled"
  objectClassRelevanceAdvisory="Enabled"
  interactionRelevanceAdvisory="Enabled"
  serviceReporting="Disabled" />

<dataTypes>

  <basicDataRepresentations>

    <basicData name="HLAinteger16BE"
      size="16"
      interpretation="Integer in the range [-2^15, 2^15 1]"
      endian="Big"
      encoding="16-bit two's complement signed integer.
      The most significant bit contains the sign." />

   <basicData name="HLAinteger32BE"
     size="32"
     interpretation="Integer in the range [-2^31, 2^31 1]"
     endian="Big"
     encoding="32-bit two's complement signed integer.
     The most significant bit contains the sign." />

   <basicData name="HLAinteger64BE"
```

```
  size="64"
  interpretation="Integer in the range [-2^63, 2^63 1]"
  endian="Big"
  encoding="64-bit two's complement signed integer first.
  The most significant bit contains the sign." />

<basicData name="HLAfloat32BE"
  size="32"
  interpretation="Single-precision floating point number"
  endian="Big"
  encoding="32-bit IEEE normalized single-precision format.
  See IEEE Std 754-1985" />

<basicData name="HLAfloat64BE"
  size="64"
  interpretation="Double-precision floating point number"
  endian="Big"
  encoding="64-bit IEEE normalized double-precision format.
  See IEEE Std 754-1985" />

<basicData name="HLAoctetPairBE"
  size="16"
  interpretation="16-bit value"
  endian="Big"
  encoding="Assumed to be portable among  hardware devices." />

<basicData name="HLAinteger16LE"
  size="16"
  interpretation="Integer in the range [-2^15, 2^15 ? 1]"
  endian="Little"
  encoding="16-bit two's complement signed integer.
 The most significant bit contains the sign." />

<basicData name="HLAinteger32LE"
  size="32"
  interpretation="Integer in the range [-2^31, 2^31 ? 1]"
  endian="Little"
  encoding="32-bit two's complement signed integer.
  The most significant bit contains the sign." />

<basicData name="HLAinteger64LE"
  size="64"
  interpretation="Integer in the range [-2^63, 2^63 ? 1]"
  endian="Little"
  encoding="64-bit two's complement signed integer first.
  The most significant bit contains the sign." />

<basicData name="HLAfloat32LE"
  size="32"
  interpretation="Single-precision floating point number"
  endian="Little"
```

```
     encoding="32-bit IEEE normalized single-precision format.
     See IEEE Std 754-1985" />

   <basicData name="HLAfloat64LE"
     size="64"
     interpretation="Double-precision floating point number"
     endian="Little"
     encoding="64-bit IEEE normalized double-precision format.
     See IEEE Std 754-1985" />

   <basicData name="HLAoctetPairLE"
     size="16"
     interpretation="16-bit value"
     endian="Little"
     encoding="Assumed to be portable among  hardware devices." />

   <basicData name="HLAOctet"
     size="8"
     interpretation="8-bit value"
     endian="Big"
     encoding="Assumed to be portable among hardware devices." />

</basicDataRepresentations>

<simpleDataTypes>

  <simpleData name="HLAASCIIchar"
    representation="HLAoctet"
    semantics="Standard ASCII character (see ANSI Std X3.4-1986)" />

  <simpleData name="HLAunicodeChar"
    representation="HLAoctetPairBE"
    units="NA"
    resolution="NA"
    accuracy="NA"
    semantics="Unicode UTF-16 character (see The Unicode Standard, Version 3.0)" />

  <simpleData name="HLAbyte"
    representation="HLAoctet"
    semantics="Uninterpreted 8-bit byte" />

  <simpleData name="HLAcount"
    representation="HLAinteger32BE" />

  <simpleData name="HLAseconds"
    representation="HLAinteger32BE"
    units="seconds" />

  <simpleData name="HLAmsec"
    representation="HLAinteger32BE"
    units="milliseconds" />
```

```
  <simpleData name="HLAfederateHandle"
    representation="HLAinteger32BE"
    semantics="The type of the argument to Normalize Federate Handle service." />

  <simpleData name="UserDataType"
    representation="HLAinteger32BE"
    units="furlongs" />

</simpleDataTypes>

<enumeratedDataTypes>

  <enumeratedData name="HLAboolean"
    representation="HLAinteger32BE"
    semantics="Standard boolean type">

    <enumerator name="HLAfalse"
      values="0" />

    <enumerator name="HLAtrue"
      values="1" />

  </enumeratedData>

  <enumeratedData name="HLAfederateState"
    representation="HLAinteger32BE"
    semantics="State of the federate">

    <enumerator name="ActiveFederate"
      values="1" />

    <enumerator name="FederateSaveInProgress"
      values="3" />

    <enumerator name="FederateRestoreInProgress"
      values="5" />

  </enumeratedData>

  <enumeratedData name="HLAtimeState"
    representation="HLAinteger32BE"
    semantics="State of time advancement">

    <enumerator name="TimeGranted"
      values="0" />

    <enumerator name="TimeAdvancing"
      values="1" />

  </enumeratedData>
```

```
<enumeratedData name="HLAownership"
  representation="HLAinteger32BE">

  <enumerator name="Unowned"
    values="0" />

  <enumerator name="Owned"
    values="1" />

</enumeratedData>

<enumeratedData name="HLAresignAction"
  representation="HLAinteger32BE"
  semantics="Action to be performed by RTI in conjunction with resignation">

  <enumerator name="DivestOwnership"
    values="1" />

  <enumerator name="DeleteObjectInstances"
    values="2" />

  <enumerator name="CancelPendingAcquisitions"
    values="3" />

  <enumerator name="DeleteObjectInstancesThenDivestOwnership"
    values="4" />

  <enumerator name="CancelPendingAcquisitionsThenDeleteObjectInstancesThenDivestOwnership"
    values="5" />

  <enumerator name="NoAction"
    values="6" />

</enumeratedData>

<enumeratedData name="HLAorderType"
  representation="HLAinteger32BE"
  semantics="Order type to be used for sending attributes or interactions">

  <enumerator name="Receive"
    values="0" />

  <enumerator name="TimeStamp"
    values="1" />

</enumeratedData>

<enumeratedData name="HLAswitch"
  representation="HLAinteger32BE">
```

```
    <enumerator name="Enabled"
      values="1" />

    <enumerator name="Disabled"
      values="0" />

</enumeratedData>

<enumeratedData name="HLAsynchPointStatus"
  representation="HLAinteger32BE"
  semantics="Joined federate synchronization point status">

  <enumerator name="NoActivity"
    values="0" />

  <enumerator name="AttemptingToRegisterSynchPoint"
    values="1" />

  <enumerator name="MovingToSynchPoint"
    values="2" />

  <enumerator name="WaitingForRestOfFederation"
    values="3" />

</enumeratedData>

<enumeratedData name="HLAserviceGroupName"
  representation="HLAinteger32BE"
  semantics="Service group identifier">

  <enumerator name="FederationManagement"
    values="0" />

  <enumerator name="DeclarationManagement"
    values="1" />

  <enumerator name="ObjectManagement"
    values="2" />

  <enumerator name="OwnershipManagement"
    values="3" />

  <enumerator name="TimeManagement"
    values="4" />

  <enumerator name="DataDistributionManagement"
    values="5" />

  <enumerator name="SupportServices"
    values="6" />
```

```
  </enumeratedData>

</enumeratedDataTypes>

<arrayDataTypes>

  <arrayData name="HLAASCIIstring"
    dataType="HLAASCIIchar"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="ASCII  String representation" />

  <arrayData name="HLAunicodeString"
    dataType="HLAunicodeChar"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="Unicode string representation" />

  <arrayData name="HLAopaqueData"
    dataType="HLAbyte"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="Uninterpreted sequence of bytes" />

  <arrayData name="HLAhandle"
    dataType="HLAbyte"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="Encoded value of a handle. The encoding is based
   on the type of handle" />

  <arrayData name="HLAtransportationName"
    dataType="HLAunicodeChar"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="String whose legal value shall be a name from any
    row in the OMT transportation table (IEEE Std 1516.2-2000" />

  <arrayData name="HLAlogicalTime"
    dataType="HLAbyte"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="An encoded logical time. An empty array shall
   indicate that the values is not defined" />

  <arrayData name="HLAtimeInterval"
    dataType="HLAbyte"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="An encoded logical time interval. An empty array
    shall indicate that the values is not defined" />
```

```
  <arrayData name="HLAhandleList"
    dataType="HLAhandle"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="List of encoded handles" />

  <arrayData name="HLAinteractionSubList"
    dataType="HLAinteractionSubscription"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="List of interaction subscription indicators" />

  <arrayData name="HLAargumentList"
    dataType="HLAunicodeString"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="List of arguments" />

  <arrayData name="HLAobjectClassBasedCounts"
    dataType="HLAobjectClassBasedCount"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="Counts of various items based on object class" />

  <arrayData name="HLAinteractionCounts"
    dataType="HLAinteractionCount"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="List of interaction counts" />

  <arrayData name="HLAsynchPointList"
    dataType="HLAunicodeString"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="List of names of synchronization points" />

  <arrayData name="HLAsynchPointFederateList"
    dataType="HLAsynchPointFederate"
    cardinality="Dynamic"
    encoding="HLAvariableArray"
    semantics="List of joined federates and the synchronization status of each" />

</arrayDataTypes>

<fixedRecordDataTypes>

    <fixedRecordData name="HLAinteractionSubscription"
      encoding="HLAfixedRecord"
      semantics="Interaction subscription information">
```

```
    <field name="HLAinteractionClass"
      dataType="HLAhandle"
      semantics="Encoded interaction class handle" />

    <field name="HLAactive"
      dataType="HLAboolean"
      semantics="Whether subscription is active (HLAtrue=active)" />

</fixedRecordData>

<fixedRecordData name="HLAobjectClassBasedCount"
  encoding="HLAfixedRecord"
  semantics="Object class and count of associated items">

   <field name="HLAobjectClass"
     dataType="HLAhandle"
     semantics="Encoded object class handle" />

   <field name="HLAcount"
     dataType="HLAcount"
     semantics="Number of items" />

</fixedRecordData>

<fixedRecordData name="HLAinteractionCount"
  encoding="HLAfixedRecord"
  semantics="Count of interactions of a class">

   <field name="HLAinteractionClass"
     dataType="HLAhandle"
     semantics="Encoded interaction class handle" />

   <field name="HLAinteractionCount"
     dataType="HLAcount"
     semantics="Number of interactions" />

</fixedRecordData>

<fixedRecordData name="HLAsynchPointFederate"
  encoding="HLAfixedRecord"
  semantics="A particular joined federate and its synchronization point status">

   <field name="HLAfederate"
     dataType="HLAhandle"
     semantics="Encoded joined federate handle" />

   <field name="HLAfederateSynchStatus"
     dataType="HLAsynchPointStatus"
     semantics="Synchronization status of the particular joined\ federate" />

</fixedRecordData>
```

```
</fixedRecordDataTypes>

</dataTypes>

 <notes>

  <note name="MOM1" semantics="The value of the Dimension Upper Bound entry for the
    Federate dimension is RTI implementation dependent." />

 </notes>

</objectModel>
```

Bibliografía

ACIMS (2004). «DEVSJava». *Informe técnico*, Arizona, center for integrative model and simulation, ACIMS.

Allen, J. (1984). «Maintaining knowledge about temporal intervals». En: *Communications of the ACM*, volumen 2, pp. 832–843.

Anderson, D. (2002). «Interview: New Supply chain capabilities». *Achieving supply chain excellence through technology*, **4**, pp. 42–43.

Arrazola, J. (2007). «Towards a new model: the globally integrated enterprise». *Informe técnico*, Universia business review.

Bakos, J.Y. (2000). «From integrated enterprise to regional clusters: the changing basis of competition». *Computer in Industry*, **42**, pp. 289–298.

Ballou, R. (1999). *Business Logistics Management. Planning, Organizing and Controlling the Supply Chain. Fourth edition.* Prentice Hall. ISBN 0-13-795659-2.

Barros, F. (1996). «Dynamic structure discrete event system specification: formalism, abstract simulators and applications». En: *Transactions of the Society for computer simulation*, volumen 13, pp. 35–46.

Barros, F.; Lehmann, A.; Liggesmeyer, P.; Verbraeck, A. y Zeigler, B. (2004). «www.dagstuhl.de/04041». *Informe técnico*.

Bernus, P. y Nemes, L. (1997). «The contribution of GERAM to consensus in the area of enterprise integration». En: *ICEIMT'97*, Springer.

Biswas, S. y Narahari, Y. (2004). «Object Oriented Modeling and Decision Support for Supply Chains». *European Journal of Operational Research*, **153(2)**, pp. 704–726.

Booch, G.; Rumbaugh, J. y Jacobson, I. (1999). *The unified modeling language user guide.* Addison-Wesley.

Browne, J. y Zhang, J. (1999). «Extended and virtual enterprise - similarities and differences». *International Journal of Agile Management System*, **1**, pp. 30–36.

Brun, A.; Cavalieri, S.; Macchi, M.; Portioli-Staudacher, A. y Terzi, S. (2002). «Distributed simulation for supply chain co-ordination». En: *12Th international Working Seminar on Production Economics*, .

Buschmann, F.; Meunier, R.; Rohnert, H.; Sommerlad, P. y Stal, M. (1996). *Pattern-oriented software architecture. A system of pattern*. Addison-Wesley.

Byrne, P. y Heavey, C. (2004). «A Framework for Analysing SME Supply Chain». En: *36th Winter Simulation Conference*, pp. 1167–1175.

CACI, product company (2006). «SIMPROCESS». *Informe técnico*, CACI Product company.

Camarinaha-Matos, L. y Afsarmanesh, H. (1999). «The virtual enterprise concept». *Infraeestructures for virtual enterprise- Networking industrial enterprises*, **153**, pp. 181–199.

Camarinha-Matos, L. (2002). «Virtual organization in manufacturing: trends and Challenges». *12th International Conference of Flexible Automation and Intelligent Manufacturing*, pp. 1036–1054.

CEN (1996). «Env 12204: Advanced manufacturing technology - systems architecture - constructs for enterprise modelling». *Informe técnico*, Comité Européen de Normalisation, Brussels.

Chalmeta, R. y Grangel, R. (2003). «ARDIN extension for virtual enterprise integration». *The journal of Systems and software*, **67**, pp. 141–152.

Chandy, K. y Misra, J. (1978). «Distributed simulation: A Case Study in Design and Verification of Distributed Systems». *IEEE transaction On software Engineer*, **5**, pp. 440–452.

Chi, S (1997). «Model-based reasoning methology using the symbolic DEVS simulation». En: *Transaction of SCS*, volumen 14, pp. 141–152.

Childe, S. J. (1998). «The Extended Concept of Co-operation». *Production Planning and Control*, **9(4)**, pp. 320–327.

Cope, D.; Fayes, M.; Mollaghasemi, M. y Kaylani, A. (2007). «Supply Chain Simulation Modeling Made Easy: an Innovative Approach». En: S. Henderson; B. Biller; M. Hsieh; J. Shortle; J. Tew y R. Barton (Eds.), *Proceedings of the Winter Simulation Conference*, .

Curtis, W.; Kellner, M. y Over, J (1992). «Process Modelling». *Comunication of the ACM*, **35(9)**, pp. 75–90.

David, P. y Anderson, R. (2003). «Improving the composability of department of defense models and simulations». *Informe técnico*, National Defense research institute.

DMSO (1998). «High level architecture version 1.3». *Informe técnico*, Defense model and simulation office.

DoD (1995). «DoD Model and Simulation Master Plan». *Informe técnico*, DoD.

Fayez, M.; Rabelo, L. y Mollaghasemi, M. (2005). «Ontologies for supply chain simulation Modeling». En: M. Kuho; N. Steiger; F. Armstrong y J. Joines (Eds.), *Winter Simulation Conference*, .

Fischer, M.; Jahn, H. y Teich, T. (2004). «Optimizing the selection of partners in production networks». *Robotics and Computers-Integrated Manufacturing*, **20(6)**, pp. 593–601.

Fox, M.; Barbuceanu, M.; Gruninger, M. y Lin, J. (1998). «An Organization Ontology for Enterprise Modelling». En: M. Prietula; K. Carley y L. Gasser (Eds.), *Simulating Organizations: Computational Models of Institutions and Groups*, pp. 131–152.

Fox, M. y Grüninger, M. (1997). «On Ontologies And Enterprise Modelling». En: Springer-Verlag (Ed.), *International Conference on Enterprise Integration Modelling Technology*, .

Fox, M. y Gruninger, M. (1998). «Enterprise modeling». *AI Magazine*, pp. 109–121.

Fujimoto, R. (1988). «Lookahead in Parallel Discrete Event Simulation». En: *proceeding of the International conference on Parallel Processing*, Silver Spring, M.D. IEEE.

Fujimoto, R. (1990). «Parallel Discrete Event Simulation». *Comunication of ACM. Special issue on simulation*, **33(10)**, pp. 30–53.

Fujimoto, R. (2003). «Distributed Simulation Systems». En: *Proceeding of the Winter Simulation Conference*, .

Gamma, E.; Helm, R.; Johnson, R. y Vlissides, J. (1996). *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley.

Gan, B.; Liu, L.; Turner, J.; Cai, W.; Jain, S. y Hsu, W. (2000). «Distributed supply chain simulation across enterprise boundaries». En: J. Joines; R. Barton; K. Kang y P. Fishwick (Eds.), *Winter Simulation Conference*, volumen 2, pp. 1245–1251.

Gruber, T. (1993). «Toward Principles for the Design of Ontologies Used for Knowledge Sharing». *Informe técnico*, Knowledge Systems Laboratory, Universidad de Stanford.

Gutierrez, M. y Leone, H. (2006). «An Environment for Developing Executable Enterprise Model». En: *proceeding of the summer simulation conference*, ISBN 1-56555-307-1.

Gutierrez, M. y Leone, H. (2008). «Using distributed simulation for analyzing Enterprise Models». En: *34 CLEI Conferencia Latinoamericana en Informática*, pp. 960–969.

Gutierrez, M.; Mannarino, G. y Leone, H. (2001a). «Coordinates Workbench: an Object-oriented architecture of a tool for conceptualizing an organization». En: *XXI international conference of the chilean Computer science society. IEEE computer society*, pp. 133–142.

Gutierrez, M.; Mannarino, G. y Leone, H. (2001b). «Coordinates Workbench: una herramienta para soportar el modelado de Empresa». En: *IDEAS 2001*, .

Harel, D. (1987). «Statecharts: a visual formalism for complex systems». *Science of computer programming*, **8**, pp. 231–274.

Harel, D. y Gery, E. (1996). «Executable Object Modeling with Statecharts». En: *International Conference on Software Engineering. Proceedings of the 18th international conference on Software engineering*, IEEE Computer Society Washington, DC, USA.

Harmon, P. (2003). «An Introduction to the Supply Chain Council´s SCOR Methodology». *Informe técnico*, Business process trends.

Hu, X.; Zeigler, B. y Mittal, S. (2005). «Variable structure in DEVS component-based modeling and simulation». *Simulation*, **81(2)**, pp. 91–102.

IEEE (2000a). «HLA Federation Development and Execution Process Model». *Informe técnico*, IEEE.

IEEE (2000b). «IEEE Standard for Modeling and Simulation High Level Architecture (HLA) - Framework and Rules». *Informe técnico*, IEEE.

IEEE (2001). *IEEE Standard for modeling and simulation High Level Architecture (HLA)*. IEEE Inc. New York USA. ISBN 0-7381-2621-7.

IEEE, computer society coordinating committee. (1990). «IEEE Standard glossary of software Engineering terminology». En: *IEEE standards board*, .

Jefferson, D. y Sowizral, H. (1985). «Fast concurrent simulation using the Time Warp Mechanism». En: *Proceeding of the summer Computer simulation. Distributed Simulation conference*, .

Jian, J.; Zhang, H.; Guo, B.; Wang, K. y Chen, D. (2004). «HLA-Based Collaborative Simulation Platform for Complex Product Design». En: *Proceedings of The 8th International Conference on Computer Supported Cooperative Work in Design*, volumen 2, pp. 563–566. IEEEE.

Johannesson, P. (2007). «The Role of Business Models in Enterprise Modelling- Conceptual Modelling». *Information Systems Engineering*, pp. 123–140.

Johnson, R. y Wolf, B. (1998). *Pattern language of progam design*. Addison-Wesley.

Krasner, G. y Pope, St. (1988). «A description of the Model-View-Controller user interface paradigm in the Smalltalk-80 system».

Law, A. y Kelton, W (1991). *Simulation modeling and analysis*. ISBN 0-07-036698-5.

Lee, Y. H.; Cho, M.; Kim, S. y Y., Kim (2002). «Supply chain simulation with discret-continuous cobined modeling». *Computers and Industrial Engineering*.

Lin, Y. y Lazowska, E. (1991). «A Study of time warp rollback mechanisms». En: *ACM transaction on modeling and computer simulation*, volumen 1, pp. 51–72.

Liu, J.; Wang, W.; Chai, Y. y Liu, Y. (2004). «EASY-SC: A Supply Chain Simulation Tool». En: R .G. Ingalls; M. D. Rossetti; J. S. Smith y B. A. Peters (Eds.), *Proceedings of the Winter Simulation Conference*, pp. 1373–1378.

Mallidi, K.; Praskevopoulos, A. y Paganelli, P. (1999). «Process modelling in small-medium enterprise networks». *Computers in Industry*, **38**, pp. 149–158.

Mannarino, G.; Henning, G. y Leone, H. (1999). «Coordinates: A Framework for Enterprise Modeling. Information Infrastructure Systems for Manufacturing». En: J. Mills y F. Kimura (Eds.), *IFIP*, pp. 379–390. Kluwer Academic Publishers.

McCormack, K. y Johnson, W. (2001). *Business Process Orientation Gaining the e-business competitive advantage*. capítulo Chapter VIII Business process orientation and Suply Chain management. CRC press LLC.

Mezgar, I.; Kovacs, G. y Paganelli, P (2000). «Cooperative production planning for small and medium-sized enterprises». *International journal of production economics*, **64**, pp. 37–48.

Möller, B. y Lofstrand, B. (2007). «Use cases for the HLA evolved modular FOMs». En: *Proceeding of the Euro simulation interoperability standards organization*, .

Möller, B.; Lofstrand, B. y Karlsson, M. (2007). «An Overview of the HLA evolved modular FOM». En: *Proceeding of the Spring simulation interoperability workshop*, .

Murata, T. (1989). «Petri Nets: Properties, Analysis and Applications». En: *Proceeding of the IEEE*, 4, pp. 541–580.

Nagel, R. y Dove, R. (1991). «Twenty-First Century Manufacturing Enterprise Strategy: An Industry-Led View». *Informe técnico*, Iacocca Institute, Lehigh Univ.

Nutaro, J. y Hammonds, P. (2004). «Combining the Model/view/Control design pattern with the DEVS formalism to achieve rigor and reusability in distributed simulation». En: *The society for modeling and simulation international*, volumen 1, pp. 19–28.

Persson, F. y Araldi, M. (2007). «The development of a dynamic supply chain analysis tool Integration of SCOR and discrete event simulation.» *International Journal of Production economics*.

Petit, M. (2002). «Report on the State of the Art in Enterprise Modelling». *Informe técnico*, UEML proyect.

Petri, C.A (1962). «Petri net, comunication with automata». *Informe técnico*, PDTIC Research Report AD0630125..

Petty, M. y Weisel, E. (2003). «A composability lexicon». En: *Proceeding of the Spring Simulation interoperability workshop*, .

Pokorny, T.; Fraser, M. y Burns, L. (2008). «portico». *software*, Australian Defense Simulation Office and Calytrix technologies.

Rice, S.; Marjanski; Markowitz y Bailey (2005). «The SIMSCRIPT III Programming Language for modular object-oriented Simulation». En: *Winter simulation conference*, .

Sarma, R.; Nidumolu, A.; Nirup, M.; Menon, B. y Zeigler, B. (1998). «Object-oriented business process modeling and simulation: A discrete event system specification framework». *Simulation practice and theory*, **6**, pp. 533–571.

Scheer, W. (1992). *Architecture of Integrated Information Systems - Bases for Company Modeling*. 2da edición.

Scrudder, R.; Lightner, G.; Lutz, R.; Saunders, R.; Little, R.; Morse, K. y Möller, B. (2005). «Evolving the High level architecture for modeling and simulation». En: *Proceeding of the 2005 interservice/industry training, simulation and education conference*, .

Shaw, M. y Garland, D. (1996). *Software Architecture. Perspectives on an emerging discipline*. Prentise hall. ISBN 0-13-182957-2.

Siegel, J. (2001). «Developing in OMF's model-drive architecture». *Informe técnico*, Object management group.

SISO-STD-003-2006 (2006). «Base Object Model (BOM) template specification». *Informe técnico*, Simulation Interoperability Standards organization.

Taylor, S.; Mustafee, N.; Turner, S.; Low, M.; Strassburger, S. y Ladbrook, J. (2007). «The SISO CSPI PDG Standard for Commercial Off-The-Shelf Simulation Package Interoperability Reference Models». En: S. Henderson; B. Biller; M. Hsieh; J. Shortle; J. Tew y R. Barton (Eds.), *Winter Simulation Conference*, pp. 594–602.

Terzi, S. y Cavalieri, S. (2004). «Simulation in the supply chain context: a survey». *Computer in Industry*, **53**, pp. 3–16.

Verbraeck, A. (2004). «Component-based Distributed Simulations. The way forward?» En: *Proceeding of the eighteenth workshop on parallel and distributed simulation*, .

Verbraeck, A.; Saanen, Y.; Stojanovic, Z.; Shishkov, B.; Meijer, A.; Valentin, E. y van der Meer, K. (2002). *Chapter 2:What are building blocks? Building blocks for effective telematics application development and evaluation*. TU delf press.

Verbraeck, A. y Valentin, E. (2008). «Design guidelines for simulation building blocks». En: S. Mason; R. Hill; L. Monch; T. Jefferson y J. Fowler (Eds.), *Proceeding of the Winter simulation conference*, .

Vernadat, F. (2002). «UEML: Towards a unified enterprise modelling language». *International Journal of production reseaarch*, **40**, pp. 4309–4321.

Vernadat, F. B. (1996). *Enterprise modeling and integration: principles and applications*. Chapman and Hall. ISBN 0-412-60550-3.

Villa (2001). «Emerging trends in large-scale supply chain management». En: *ICPR 16th international conference on production research*, .

VonMevius, M. y Pibernik, R. (2004). «Process Management in Supply Chain - A New Petri Net Based Approach». En: *Proceedings of the 37th Hawaii International Conference on System Sciences*, .

Warmer, J. y Kleppe, A. (2003). *The Object Constraint Language. Second Edition. Getting your models ready for MDA*. Addison-wesley.

White, T. y Barnett, M. (2003). «Driving supply-chain policy decisions using scor-based simulation». En: *APICS International conference and exposition*, .

Wu, N. y Su, P. (2005). «Selection of partners in virtual enterprise paradigm». *Robotics and Computer-Integrated Manufacturing*, **21(2)**, pp. 119–131.

XML, Core working group. (2008). «XML 1.1 (fifth edition)». *Informe técnico*, W3C.

Zeigler, B. (1976). *Theory of Modelling and simulation*. Wiley. ISBN 4-71981-52-4.

Zeigler, B.; Ball, G.; H., Cho y H., Sarjoughian (1998). «The DEVS/HLA distributed simulation environment and its support for predictive filtering». En: *Advance simulation technology Thrust (ASTT)*, .

Zeigler, B.; Herbert, P. y Tag Gon, K. (2000a). *Theory of Modeling and Simulation. Integrating Discrete event and continuous complex dynamic systems*. capítulo DEVS-based Extended Formalisms, pp. 252–258. Academic press.

Zeigler, B.; Kim, T. y Praehofer, H. (2000b). *Theory of modelling and simulation. Integrating discrete event and continuouis complex dynamic systems.* Acadamic Press, second[a] edición. ISBN 0-12-778455-1.

Zeigler, B.; Moon, Y.; Kim, D. y Kim, J. (1996). «DEVS-C++: A High Performance Modelling and Simulation Environment». En: *Proceedings of 29t. Hawaii International Conference on System Sciences*, .